SEVENTH FRAMEWORK PROGRAMME
Trustworthy ICT

Project Title:

Enhanced Network Security for Seamless Service Provisioning in the Smart Mobile Ecosystem

Grant Agreement No: 317888, Specific Targeted Research Project (STREP)

DELIVERABLE

D2.2 Honeydroid: Virtualized Mobile Honeypot for Android

Deliverable No.		**D2.2**	
Workpackage No.	**WP2**	Workpackage Title	**Development of Virtualized Honeypots for Mobile Devices**
Task No.	**T2.2**	Task Title	**Virtualized Mobile Honeypot Development**
Lead Beneficiary		**TUB**	
Dissemination Level		**PU+RE**	
Nature of Deliverable		**R+P**	
Delivery Date		**14-11-2014**	
Revision Date			
Status		**F**	
File name		**NEMESYS_Deliverable_D2.2.pdf**	
Project Start Date		**01 November 2012**	
Project Duration		**36 Months**	

Authors List

Author's Name	Partner	Email Address
Lead Author / Editor		
Janis Danisevkis	TUB	janis@sec.t-labs.tu-berlin.de
Co-Authors		
Bhargava Shastry	TUB	bshastry@sec.t-labs.tu-berlin.de
Kashyap Thimmaraju	TUB	kashyap.thimmaraju@sec.t-labs.tu-berlin.de
Matthias Petschick	TUB	matthias@sec.t-labs.tu-berlin.de
Steffen Liebergeld	TUB	steffen@sec.t-labs.tu-berlin.de
Matthias Lange	TUB	mlange@sec.t-labs.tu-berlin.de
Dario Lombardo	Telecom Italia	dario.lombardo@it.telecomitalia.it

Reviewers List

Reviewer's Name	Partner	Email Address
Mihajlo Pavloski	ICL	m.pavloski13@imperial.ac.uk
Laurent Delosieres	HIS	ldelosieres@hispasec.com

Contents

List of Figures **7**

List of Tables **8**

1 Introduction **12**
1.1 Deliverable Overview . . . 13
1.2 Organization . . . 13

2 Background **15**
2.1 Honeypots . . . 15
2.1.1 High-interaction Vs. Low-interaction . . . 15
2.1.2 Physical Vs. Virtual . . . 16
2.1.3 Server Vs. Client . . . 16
2.2 Operating System Concepts . . . 17
2.2.1 Privilege Levels . . . 17
2.2.2 Processes and Tasks . . . 17
2.3 Micro-kernel . . . 17
2.4 Android Open Source Project . . . 18
2.5 Fiasco.OC Micro-kernel . . . 18
2.6 L4Linux . . . 19

3 Related Work **20**
3.1 Virtualized Honeypots . . . 20
3.1.1 Xebek . . . 20
3.1.2 Argos . . . 21
3.1.3 Potempkin . . . 21
3.2 Non-virtualized Honeypots . . . 21
3.2.1 HosTaGe . . . 21
3.2.2 Sebek . . . 22
3.3 Hybrid Systems . . . 22
3.3.1 ReVirt . . . 22
3.3.2 DroidScope . . . 22

3.3.3 CopperDroid . . . 22

4 Honeydroid: Requirements Specification 24
4.1 Visibility . . . 24
4.2 Integrity of Audit Logs . . . 24
4.2.1 System Call Meta-data . . . 25
4.2.2 Disk Snapshots . . . 25
4.3 Containment . . . 26
4.4 Exposure . . . 26

5 Honeydroid: Design and Implementation 28
5.1 Software Architecture . . . 28
5.1.1 Server . . . 28
5.1.2 Compartment . . . 29
5.2 Platform Control . . . 30
5.3 Networking . . . 30
5.3.1 Modem . . . 31
5.4 Input . . . 31
5.5 Output . . . 32
5.5.1 Graphics Stack . . . 32
5.5.2 Audio subsystem . . . 32
5.6 Mass Storage . . . 33
5.7 Anomaly Sensors . . . 34
5.7.1 System call sensor . . . 34
5.7.2 Screen state sensor . . . 35
5.8 Forensic Data Acquisition . . . 36
5.9 Premium SMS Filter . . . 36
5.9.1 SMS Routing . . . 36
5.9.2 Content Filtering . . . 37

6 Communication with Service Provider 39
6.1 Introduction . . . 39
6.1.1 Anomaly detection message from the mobile Honeydroid to the collector . . . 39
6.1.2 Response message from the collector to the mobile Honeydroid . . . 40

7 Evaluation 42
7.1 Classification of Performance and Benchmarks . . . 42

7.1.1 Classification of Performance . . . 42
7.1.2 Classification of Benchmarks . . . 42
7.2 Methodology . . . 44
7.3 Evaluation Framework . . . 45
7.4 Experimental Setup . . . 45
7.5 Network Evaluation . . . 48
7.5.1 Iperf 2 . . . 48
7.5.2 Honeydroid and Vanilla Down-link Tests . . . 50
7.6 Operating System Evaluation . . . 51
7.6.1 LMbench . . . 51
7.6.2 Hackbench . . . 56
7.7 CPU Evaluation . . . 57
7.7.1 LINPACK . . . 58
7.7.2 Scimark2 . . . 59
7.7.3 Real Pi . . . 60
7.8 Application Evaluation . . . 61
7.8.1 Dalvik VM garbage collection . . . 61
7.8.2 Sunspider . . . 62
7.9 Power Evaluation . . . 64
7.9.1 Idle Test and execution . . . 64
7.9.2 Voice Call Test and execution . . . 64

8 Evaluation Results **65**
8.1 Performance Results and Analyses . . . 65
8.2 Networking . . . 65
8.2.1 Results . . . 65
8.2.2 Analysis . . . 66
8.3 Operating System Results . . . 69
8.3.1 LMbench . . . 69
8.3.2 Hackbench . . . 73
8.4 CPU Results . . . 75
8.4.1 LINPACK . . . 75
8.4.2 Scimark2 . . . 75
8.4.3 RealPi Bench . . . 77
8.5 Application Results . . . 79
8.5.1 Dalvik VM Garbage Collector . . . 79
8.5.2 Sunspider . . . 81
8.6 Power Results . . . 85

8.6.1 Idle state Results 85
8.6.2 Voice Call Results 86
8.6.3 Analysis 86

9 Conclusion **88**

List of Figures

5.1 Honeydroid: Architectural overview 29
5.2 Goos protocol primitives. 33
5.3 nbd setup 34
5.4 35

7.1 A block diagram of the performance evaluation framework 46
7.2 A photograph of the experimental setup 47

8.1 Vanilla and Honeydroid comparison of Network performance with MTU=604B 68
8.2 Vanilla and Honeydroid comparison with MTU=1500B 68
8.3 Vanilla and Honeydroid comparison of system calls latency 70
8.4 Vanilla and Honeydroid comparison of system calls bandwidth 71
8.5 Vanilla and Honeydroid comparison of Hackbench Sockets and Pipes . . . 74
8.6 Vanilla and Honeydroid comparison for LINPACK 76
8.7 Vanilla and Honeydroid comparison 78
8.8 Vanilla and Honeydroid comparison of Dalviks VM garbage collector . . . 80
8.9 Vanilla and Honeydroid comparison for Sunspider 84
8.10 Idle test results for Vanilla and Honeydroid 85
8.11 Voice call results 86

List of Tables

8.1 TCP Throughput of Vanilla and Honeydroid with MTU=604 bytes 66
8.2 TCP Throughput of Vanilla and Honeydroid with MTU=1500 bytes . . . 67
8.3 UDP Throughput of Vanilla and Honeydroid with MTU=1500 bytes . . . 67
8.4 UDP Throughput of Vanilla and Honeydroid with MTU=1500 bytes . . . 67
8.5 System call latency of Vanilla and Honeydroid 69
8.6 Memory bandwidth of Vanilla and Honeydroid in moving 10.49MB 70
8.7 Hackbench performance of Vanilla and Honeydroid 73
8.8 LINPACK performance of Vanilla and Honeydroid 75
8.9 Scimark2 performance of Vanilla and Honeydroid 77
8.10 RealPi Bench performance of Vanilla and Honeydroid 77
8.11 Dalvik VM Garbage collection performance of Vanilla and Honeydroid . . 79
8.12 Sunspider results for Vanilla 82
8.13 Sunspider results for Honeydroid 83

Abbreviations

2G	Second Generation
3G	Third Generation
ADB	Android Debug Bridge
AMD	Advanced Micro Devices, Inc.
AOSP	Android Open Source Project
ARM	Advanced RISC Machines Ltd.
BSP	Board Support Package
BYOD	Bring Your Own Device
CPU	Central Processing Unit
CSS	Cascaded Style Sheet
DAC	Discretionary Access Control
DEF	Dalvik Executable Format
DMA	Direct Memory Access
GID	Group IDentifier
HTC	High Tech Computer Corporation
HTML	Hyper Text Markup Language
IMEI	International Mobile Equipment Identity
IP	Internet Protocol
JTAG	Joint Test Action Group
LLB	Low Level Bootloader
MAC	Mandatory Access Control
MMS	Multimedia Messaging Service
MVP	Mobile Virtualization Platform
MWR	MWR InfoSecurity
NX	No Execute
OK	Open Kernel Labs
PIE	Position Independent Executable
QNX	QNX micorkernel
RAM	Random Access Memory
RIM	Reseach In Motion
ROM	Read Only Memory
ROP	Return Oriented Programming
TI	Texas Instruments Inc.
UDID	Unique Device IDentifier
UID	Unique IDentifier
USB	Universal Serial Bus
XD	eXecute Disable
XN	eXecute Never
XNU	X is Not Unix

Abstract

A surge in the popularity of smartphones has seen a seemingly parallel surge in attacks targeting them. Given that smartphones are private devices with a potential for valuable information, attacks against them have become more and more sophisticated. In this report, we argue that the honeypot concept is relevant to end-user devices. We make a first attempt at realizing a virtualized smartphone honeypot called Honeydroid.

Honeydroid is a virtualized smartphone honeypot based on the Android OS and targeted at Samsung Galaxy S2 devices. Virtualization facilities are provided by the Fiasco micro-kernel. In this report, we document the design and implementation of the developed prototype. Subsequently, we benchmark the developed prototype with regard to its power consumption, and CPU and network performance.

1 Introduction

Smartphones have grown significantly every since their arrival. A Gartner study from July 2014 [32] shows that smartphone sales continue to dominate the consumer electronic device market, and that the share of smartphones in global mobile device sales is expected to reach 88 percent in 2018, up from 66 percent in 2014. This shift in mobile usage has attracted malicious actors like malware authors and cyber criminals. Recent research has shed light on multiple aspects of the modus operandi of adversaries targeting Android smartphones [55]. This motivates the idea of a systematic framework to gather information relating to the deployment and operation of malware. Honeydroid is an effort to advance research in this direction.

Classically, honeypots have been used in the networking infrastructure to serve as an early alarm for intrusions. Furthermore, honeypots have been constructed as shadow instances of real services. This meant that a compromised honeypot is harmless since they were not involved in service delivery. This model of detecting intrusions works well for attacks targeting a network; however, porting this model to smartphones is challenging.

Adversaries who target smartphones have a larger attack surface at their disposal. For instance, in delivering a malicious payload which is the first step of an attack, the adversary may use multiple channels. The malicious payload may be delivered over WiFi, bluetooth, or Near-Field Communication (NFC) interface, packaged as a smartphone application on a smartphone application market etc. Once delivered, the malicious payload executes instructions that cause harm to the user and optionally attempt to elevate their privilege level so as to persist on the device and hide themselves from anti-malware applications. In order to capture this multi-stage modus operandi of malware, the entire smartphone operating system stack needs to be instrumented as a honeypot allowing us to detect intrusions launched by various means and to monitor their behavior on the device.

In this report, we describe, how the Android OS stack has been instrumented to develop Honeydroid: a virtualized smartphone honeypot. We leverage isolation offered by a micro-kernel OS called Fiasco.OC to create two virtual compartments: honeypot Virtual Machine (VM) and monitoring VM. We implement logging or auditing mechanisms in the monitoring VM thereby isolating it from the honeypot compartment. The isolation enforced by the micro-kernel allows us to maintain integrity of log data. The log

data may be used offline by a security analyst to investigate the root-cause of a malware incident.

Apart from the core design and implementation of Honeydroid, we discuss forensics and auditing features such as disk snapshots, system call histogram logging, and a rudimentary premium SMS filter. While disk snapshots are meant to aid a forensic analyst in pin-pointing the root cause of an attack, system call histogram is used by the Lightweight Malware Detector (LMD) module (Deliverable 2.3) in anomaly detection. The SMS filter is designed to stop outgoing (fraudulent) premium SMS messages that is a common monetization mechanism among Android malware.

Finally, we describe a methodology to evaluate the overhead of the developed honeypot prototype and apply it to our setup. The evaluation results show that Honeydroid incurs a modest performance penalty for Android and Java benchmarks and that its network performance on a 3G/GPRS network is on par with vanilla (or unmodified) Android phones.

1.1 Deliverable Overview

This deliverable (D2.2) is part of the NEMESYS Work Package 2 (WP2) that is concerned with the development of a virtualized smartphone honeypot. It documents work carried out as part of realizing an Android-based virtualized honeypot called Honeydroid. Deliverable 2.3 titled "Lightweight Malware Detector" describes the work carried out in implementing an anomaly detection module on the smartphone device that is capable of flagging malicious applications to the end-user. Prior to this work, a survey was carried out in D2.1 [37] in order to select a development platform for a virtualized smartphone honeypot. The survey indicated that Android OS was the most suited for prototyping the honeypot framework within the contours of the Android Open Source Project (AOSP) [1].

1.2 Organization

The rest of this document is organized as follows: Chapter 2 presents the reader with background information essential to understand key operating system and security concepts. Chapter 3 presents the state-of-the-art in virtualized and non-virtualized honeypot research and places our work in context. Chapter 4 sets out key scientific requirements for a smartphone honeypot. Chapter 5 describes the design and implementation of Honeydroid. Chapter 6 describes the design and implementation of the communication protocol between the monitoring Virtual Machine (VM) in Honeydroid and the

telecommunication service provider. Chapters 7 8 discuss methodology employed to evaluate Honeydroid's performance and results gathered in the evaluation study respectively. Chapter 9 concludes this report.

2 Background

2.1 Honeypots

In the realm of computer science honeypots are computer systems meant to be compromised or probed. A more precise definition of a honeypot is "an information system resource whose value lies in unauthorized or illicit use of that resource"[46]. Honeypots by definition are supposed to have no production value therefore any access to the Honeypot can be deemed suspicious. Honeypots are widely deployed in production networks to lure attackers and understand the attackers intent or behavior. They are particularly useful in understanding zero day attacks or programs intending to gain privileged information or access.

So far, honeypots have mainly resided on the network as physical or virtual honeypots in the form of routers, firewalls, switches or PCs. This has proven useful in a number of ways such as identifying anomalous behavior, worm propagation and developing signatures for Intrusion Detection Systems(IDS), Spam filtering and Intrusion Prevention System(IPS).

A honeypot may be a virtual or physical system, it may have a high level of interaction or low level of interaction or it may be a server or a client. Since certain malicious programs these days are intelligent enough to not misbehave in a virtualized environment[44]. Therefore in order to deploy a honeypot on the production network, the requirements and capabilities of the honeypot in question must be carefully understood. The following subsection will highlight the differences between the different types of honeypots prevalent today.

2.1.1 High-interaction Vs. Low-interaction

High-interaction honeypots are considered to be real physical systems with an Operating system offering native services. Since a high-interaction honeypot is running a full suite of daemons and applications and its associated vulnerabilities it has a better chance of enticing malware or unauthorized access. What is particularly interesting about high-interaction honeypots is the concept of a honeynet. A honeynet is a network of honeypots, all network and end devices are honeypots. Such a network can be used to study Advanced persistent threats, malware propagation and subtle attacks

that would go unnoticed otherwise. The downside to using high-interaction honeypots is that they are expensive to maintain, entail high risk due to their high-interaction and time consuming analysis of security incidents. A Virtual Machine can be considered a high-interaction honeypot even though its not a physical system.

Low-interaction honeypots on the other hand are honeypots that do not offer a full range of services. Honeyd [14], for instance, is a honeypot framework that instantiates only a subset of services found on a real network. In contrast to high-interaction honeypots, these are easier to maintain. On the other hand, due to a limited number of services available, low-interaction honeypots may be of lesser value in gathering malware reconnaissance.

2.1.2 Physical Vs. Virtual

Physical honeypots are very similar to high-interaction honeypots, they are considered to be physical devices running a desired operating system. The complete honeypot may be compromised by the attacker which makes this kind of a honeypot expensive and nearly impossible to scale as maintenance becomes extremely difficult and cumbersome.

Virtual honeypots on the other hand make use of Virtual Machines. This allows the possibility to scale, move and modify the honeypot almost instantly. Virtual honeypots can be high-interaction or low-interaction honeypots but the main point here is that the honeypot is hosted on a virtual machine. The virtualization of the honeypot can be accomplished by commercial hardware and software offered by VMware or Virtual Box or by open source solutions such as User-Mode Linux or QEMU. This method allows the user flexibility, scalability and portability of the honeypot/s unlike physical honeypots. These honeypots may fail to trick intelligent malware or attackers in some cases.

2.1.3 Server Vs. Client

Server honeypots are the honeypots that have been discussed until now. Client honeypots are honeypots that are deployed on the client side[50]. They are meant to interact with malicious servers for e.g a client honeypot could use a web browser to visit a lit of web-pages that are hosted on possible malicious web servers. The honeypot logs the behavior of the web-server with its client which can then be analyzed to identify malicious web servers, classify server or client based attacks. There are certain nuances to client honeypots, the honeypot makes the first move in being compromised and the false positive number is much higher than server honeypots.

2.2 Operating System Concepts

For the benefit of the reader, operating system concepts that are referred to in the rest of this document are briefly defined in this section.

2.2.1 Privilege Levels

Modern CPUs support two or more privilege levels to allow for resource management, fault isolation and security. Linux makes use of two privilege levels: supervisor mode and user mode.

2.2.2 Processes and Tasks

Processes and tasks are units of isolation in an operating system. There are multiple aspects of isolation: spatial isolation via virtual address spaces, and resource management via file descriptors or capabilities.

Spatial Isolation Modern CPUs allow to construct virtual address spaces whereby access to physical memory can be both confined and redirected. An activity executing in a virtual address space can only access memory that has been associated with that address space. Mutually distrusting activities are assigned to different address spaces.

Resource Management Modern operating systems provide mechanisms to bind resources to processes or tasks. File descriptors have names local to a process or task and refer to a resource e.g., file, device.

2.3 Micro-kernel

Micro-kernels provide only the most essential OS services to user space applications. These include inter-process communication (IPC), spatial isolation, and scheduling. Only these services run in the supervisor mode. Since the lines of code are reduced by several orders of magnitude, it allows for better auditing of software and fewer bugs. Non-essential services, like device drivers, memory management, runtime libraries etc., are delegated to be run in user mode.

2.4 Android Open Source Project

The Android Open Source Project (AOSP) [1] refers to the publicly available implementation of the Android operating system. AOSP comprises of 3 layers in the Android OS software stack, namely:

1. Linux kernel
2. Middleware modules
3. Default applications

The Linux kernel provides essential operating system services like process scheduling, a filesystem etc. Apart from core OS services, the kernel provides an API for hardware devices such as the WiFi chip, bluetooth, telephone modem etc. Android's middleware modules written in C++ and Java provide another layer of abstraction over the Linux kernel. For instance, Android middleware comprises of an interpreter for application byte code, and provides services to manage applications (apps), make use of other application services via Inter-Process Communication (IPC) and so on.

The middleware module expose a Java-written API to Android applications. Given that the Java APIs abstract hardware and software services, most Android apps are written in Java. Application runtime on Android consists of the Dalvik Virtual Machine (VM). Android applications are compiled into a byte code format known as the Dalvik executable (Dex). The Dalvik VM interprets Dex byte code at runtime.

Although a majority of Android apps are written in Java and compiled to Dex, AOSP permits applications to make use of Linux kernel APIs like open(), exec() etc. Such applications are colloquially called "native" apps because the application code is compiled to machine code and hence is *native to the underlying processor*.

2.5 Fiasco.OC Micro-kernel

Fiasco.OC (Object Capability) [38, 40, 34] is a micro-kernel in the tradition of L4 μ-kernels. It features scheduling, spacial isolation and mechanisms for inter process communication. Fiasco.OC also features a capability based access model, that is all kernel objects can only be accessed through capabilities. Capabilities are referred to by local identifiers that are only meaningful within a certain protection domain (much like file-descriptors on UNIXoid OSes). Fiasco.OC provides the following abstraction as kernel-objects that can be referred to using capabilities [3]:

Task Comprises a virtual address space as well as a collection of access rights, capabilities. Thus a task represents a protection domain.

Thread Schedulable entity associated with a task that it executes in.

IPC-Gate Synchronous cross task communication object. Owners of a capability can invoke (client-side) or attach to (server-side) to an IPC-Gate. The identity of an IPC-Gate is not forgeable.

IRQ Asynchronous cross task communication object. Owners of a capability can trigger (event-source) or attach to (event-sink) an IRQ. Triggering an IRQ is always non-blocking and its identity is not forgeable.

Factory Creates new object of the above types as well as new factory objects thereby restricting kernel quota.

Scheduler Allows setting of scheduling parameters of a thread such as CPU-affinity and priority. This object is a Singleton.

ICU The interrupt control unit represents all physical interrupt sources in the platform and allows clients to bind IRQ-object to these sources.

VLog This primitive debugging features allows to send output to a serial line.

With these primitives it is possible to build very dynamic systems in the user-space of the Fiasco.OC kernel as well as re-hosting operating systems such as Linux. The re-hosting is aided by the VCPU mode of threads which allows to process events (interrupts) and exceptions (page-faults, access violations, system calls) asynchronously as well as the transition to secondary tasks.

2.6 L4Linux

L4Linux is the re-hosted version of Linux that is capable of running on top of Fiasco.OC. It uses tasks as primitives for spacial isolation for its processes and kernel. One or more threads in VCPU mode, typically one per physical CPU, are used for scheduling. The guest kernel freely distributes the CPU-time it receives per VCPU among its threads. For Honeydroid L4Linux has been patched with the additions Google made to Linux. It provides all the services needed by the Android middleware and is therefore capable of supporting the Android middleware and its applications.

3 Related Work

In this section, we describe related work on Honeypot Development. We structure the section on a key design parameter—whether to have: (1) virtualized honeypot i.e., honeypots that can be dynamically scheduled whereby multiple virtual honeypots on run on a single physical machine; (2) physical honeypots i.e., honeypot systems that require an entire physical host for its operation; or (3) A hybrid system that permits analysis of captured data in addition to serving as a stand-alone honeypot.

3.1 Virtualized Honeypots

3.1.1 Xebek

Xebek [47] is a virtualized honeypot framework based on the Xen Virtual Machine Monitor (VMM). Xebek was primarily designed to address design issues with an earlier honeypot instrumentation tool called Sebek (See Section 3.2.2). While Sebek is designed to capture events triggered by a potential attacker (e.g., keystrokes, browsing activity etc.), its presence can be detected with moderate effort on the part of the attacker [1]. This can then be used to circumvent information gathering itself making the honeypot ineffective.

Xebek attempts to address the problem of covert information gathering by the following design decisions that it takes:

1. Patches system-calls in the guest Operating System (OS) instead of running as a kernel module

2. Not use the network stack of the guest OS to send out gathered traces. Instead use inter-VM shared memory for copying traces to domain 0 (dom 0)

3. Move the central logging server to Dom0 instead of exposing it to the network.

[1] Sebek is implemented as a hidden kernel module

3.1.2 Argos

Argos [45] is a hosted hypervisor based honeypot framework that, in addition to gathering information about intrusions, claims to contain the damage arising out of a compromised system. Argos is a QEMU-based framework.

3.1.3 Potempkin

Potempkin [53] strives to achieve an acceptable trade-off with regard to high fidelity—gathering as much information about the attacker as possible—and high scalability—the ability to monitor a large number of Internet hosts. In terms of design and implementation, Potempkin can be described as a network of honeypots—called a honeyfarm by the authors—that are implemented as a bunch of Xen VMs.

The choice of having VMs instantiate honeypots is driven by the manageability that VMs offer in terms of being well suited for booting up/bringing down VMs on demand.

3.2 Non-virtualized Honeypots

3.2.1 HosTaGe

HosTaGe [52] is, as the paper suggests, a low interaction honeypot for smartphones. HosTaGe is developed as an Android application that is designed to detect malicious network intrusions. While, at the outset, HosTaGe and Honeydroid seem similar in terms of implementing a functional honeypot on a mobile device, there are multiple differences, outlined in the following paragraphs, that set the two apart.

HosTaGe is a low-interaction honeypot that hosts shadow instances of essential networking services such as ftp, http, ssh, and so on. On the other hand, Honeydroid is a high-interaction honeypot that instantiates the entire mobile OS stack on top of a micro-kernel. This allows us to monitor the honeypot with high fidelity as in other solutions such as Potempkin and Xebek.

HosTaGe is developed as an Android application that is not resilient to attacks and/or being subverted. Given that the Android marketplace is ripe with malware and it is relatively simple for an adversary to disable apps selectively, not being resilient to attacks is something a mobile honeypot system cannot afford. Honeydroid isolates the honeypot VM from the monitoring VM. The isolation property of Honeydroid is derived from the Fiasco.OC micro-kernel. Therefore, a compromised Honeypot VM will continue to be monitored as intended and provide insights into an attacker's modus operandi.

3.2.2 Sebek

Sebek [5] is an data gathering tool that is implemented as a hidden kernel module and available for Unix and Windows OSs. It is designed to stealthily gather attack traces from a potential attacker on a given system. Sebek was one of the first honeypot frameworks to be implemented in the wild as part of the honeynet project.

3.3 Hybrid Systems

3.3.1 ReVirt

ReVirt [30] is a virtualized honeypot that is monitored by the underlying virtual machine monitor extensively. The design enables logging of non-deterministic events as well as the ability to record-and-replay.

While ReVirt can be classified as a virtualized honeypot, it is in fact a hybrid system because, ReVirt is neither meant to be portable to multiple hardware platforms nor is it meant to be booted-up and shut-down on demand as is the case with a system like Potempkin. In other words, it is more tightly coupled to the underlying hardware than is the case with Potempkin.

Like ReVirt, Honeydroid is designed to be able to log non-deterministic events such as writes to disk. Furthermore, Honeydroid allows us to revert to a previously obtained disk snapshot of the system. This enables us to switch to a known safe state before executing a malicious program.

3.3.2 DroidScope

DroidScope [54] is a hosted hypervisor or emulator based honeypot/dynamic analysis framework for Android. The primary contribution of authors of DroidScope is that, unlike dynamic analysis frameworks for Android like Anubis [2], DroidScope reconstructs the state of the Android runtime i.e, Dalvik VM in addition to the kernel-level state. This aids the security researcher in gathering and subsequently analyzing both high-level contextual information from Android middleware/runtime as well as from low-level kernel space information.

3.3.3 CopperDroid

CopperDroid [31] is a malware analysis framework for Android that is based on classical virtual machine introspection techniques. Like DroidScope, CopperDroid is based on the Android QEMU interposing between the emulated platform i.e., Android, and the

hosting environment i.e., a PC operating system such as Linux or Windows. Copper-Droid's key assumption is the claim that system-call behaviors of malware is sufficient to capture malicious behavior per se and therefore, monitoring system-call centric behavior is sufficient to detect and analyze multiple types of Android malware.

4 Honeydroid: Requirements Specification

The objective of this deliverable is to develop a smartphone honeypot prototype on Android OS as identified in Deliverable 2.1 [37]. Developing a virtualized honeypot is a fundamental requirement to providing an attack tolerant environment for the honeypot. Virtualization enables many of the fundamental requirements described in the rest of this chapter. Coupled with the low Trusted Computing Base (TCB) of the micro-kernel, Honeydroid's architecture ensures that its attack surface remains small.

4.1 Visibility

A honeypot needs to have visibility into the actions of a malicious program (malware). Visibility is key to understanding the instructions e.g., system calls, that the malware makes or the data it stores to disk. It is important to note that, more the information available to a security analyst from a smartphone honeypot, the easier is the task of investigating the root cause of a malware infection and possible mitigation strategies. It follows that this requirement may be framed in terms of how much visibility a honeypot retains into the internal workings of the Device Under Test (DUT)—the Android OS and applications in our case. While instrumenting a native operating system stack as a honeypot (or, having a non-virtualized honeypot architecture) theoretically provides one a very high degree of visibility, it also means that monitoring activities i.e., processes involved in gathering attack related information from the honeypot, are vulnerable to a compromise since they run within the same operating system. In such a setup, a compromise of the honeypot may result in the compromise of the monitoring activities as well. This brings us to the second key requirement i.e., integrity of audit logs.

4.2 Integrity of Audit Logs

Auditing is the process of passively gathering information. In the context of Honeydroid, auditing refers to the collection of attack related information. Since audit logs contain contextual information that aids in understanding the modus operandi of an attacker, possibly giving clues towards attack mitigation as well, it is important that the integrity of the audit logs be maintained. As mentioned in the previous section,

Honeydroid architecture should ensure that the auditing mechanisms are robust even when the Honeypot Virtual Machine (VM) is compromised. There is a tension between ensuring high visibility yet maintaining the Integrity of audit logs. The virtualized architecture of Honeydroid strives to maintain a balance between these two requirements while ensuring that integrity of audit logs is not compromised at any rate.

There are two key requirements for the auditing framework of Honeydroid. They are, providing system call meta-data to the Lightweight Malware Detector (LMD) module, and the ability to take disk snapshots. Each of these are briefly described in the following paragraphs. Please refer to Chapter 5 for a detailed description of the same.

4.2.1 System Call Meta-data

Honeydroid employs an on-device malware detection module hereafter called the Lightweight Malware Detector or LMD. LMD is designed to detect anomalies in application behavior and provide information regarding possible intrusions to the service provider. For a full description of the inner workings of LMD, the reader is directed to Deliverable 2.3 of the same work package.

The LMD requires analyses the frequency of system calls invoked by applications on the Honeypot VM towards detecting anomalies in application behavior. Thus, it is required that the Honeydroid architecture cater to this requirement. Since the LMD is located in the monitoring VM, the Honeydroid architecture needs to leverage inter-VM communication towards this end. The process of auditing system call meta-data is mediated by the Fiasco.OC micro-kernel. A full description of how this is done is provided in Chapter 5.

4.2.2 Disk Snapshots

An application on the honeypot VM makes active use of flash disk to store code and data persistently on the device. For instance, when an application is downloaded from an application market, it is installed on the disk. Subsequently, a data directory is created for the application to use. At runtime, the application writes to and reads from this data directory. It follows that application code and data form a key component of the auditing framework. Application code and data analysis provide a security analyst with a high degree of attack relevant data. Thus, it is required that there be a mechanism to take snapshots of the flash disk at different points in time so that they are available for offline analysis. Disk snapshots are thus, nothing but the frozen state of a flash disk—comprising of data and code of multiple applications—at a given point in time.

A second requirement is that the flash disk be revertable to a safe state in case of a

malware infection e.g., a soft reboot of Honeydroid should revert the operating system to a state that existed before the malware infection took place. This ensures that the system is not harmed by potentially harmful activities of a malicious application while retaining a snapshot into the inner workings of the application. This brings us to the third requirement, containment of potentially undesirable activities.

4.3 Containment

A common monetization mechanism of malicious applications is via premium SMSs. Typically, the adversary registers a premium SMS number with the service provider. Subsequently, the adversary engineers a malicious smartphone application, which when installed on to victims' devices, covertly sends SMSs to the adversary's number. Each of these SMSs amount to a small fraction of the money the adversary makes but when installed by a large number of devices through a centralized application store like Google's Playstore, the revenue accruing to the adversary is substantial.

Thus, a key requirement from Honeydroid is the ability to stop outgoing SMSs to undesirable premium SMS numbers. A use-case is the mobile network operator blacklist a set of premium SMS numbers that are known to be registered to malicious entities. In technical terms, this requires that the modem driver be housed in the monitoring compartment, so that the user retains control over sending of SMSs.

4.4 Exposure

As mentioned in Chapter 2, a honeypot's value lies in being compromised. Measuring how compromisable a honeypot is in quantitative terms is difficult. However, we can lay down ground rules to make operate a honeypot in an environment that is congenial to a potential compromise. A radical approach taken in the NEMESYS project is to work towards this goal by realizing a smartphone honeypot—a honeypot that is deployed in the hands of users. In the context of this requirement, exposure of the honeypot refers to its exposure to the outside world. The more exposed the honeypot is, the more likely it is to be attacked. By virtualizing smartphone device hardware such as the modem stack, we ensure that real-world malware infection channels are functional on the honeypot. While not a technical design objective, we intend to achieve higher visibility by handing over the virtual honeypot to users in field trails. We refer the reader to the report D 7.2.1 on the initial outcomes in this direction [43].

In summary, we have outline four key requirements for a smartphone Honeypot. In the next chapter, we describe the design and implementation of Honeydroid and in the

process throw light on how the design works towards the requirements set out.

5 Honeydroid: Design and Implementation

5.1 Software Architecture

Honeydroid was designed based on the principle of "Multiple Independent Levels of Security" (MILS) which is a high-assurance security architecture[49].

A micro-kernel such as Fiasco.OC offers system developers with a small secure kernel, secure communication channels and Object Capabilities that is highly necessitated by the MILS principles. Using such concepts has created a secure system from the bottom up which can be leveraged in developing a honeypot with non-bypassable sensors for intrusion detection.

The small code base of the micro-kernel offers a better Trusted Computing Base (TCB) and lower complexity within the code thereby leading to fewer bugs but it does entail extra work in the applications or operating systems that use the micro-kernel.

Honeydroid, being built on the basis of Fiasco.OC and L4Re, is naturally composed of the primitives introduced in Section 2.5. Figure 5.1 shows the basic architecture of Honeydroid. The Honeydroid VM is the user facing component. It comprises a full Android stack on top of an instance of L4Linux as of Section 2.6. We call this a *compartment*. For security reasons it must not access hardware devices directly. To provide the experience a user is hoping for Honeydroid has infrastructure that provides all the services needed to the Honeydroid VM. The infrastructure comprises a number of *servers* running natively on Fiasco.OC and another instance of L4Linux, the Monitoring VM or Monitoring compartment. We will now introduce the terms *server* and *compartment* in more detail, thereby explaining there role in the Honeydroid system architecture.

5.1.1 Server

The term L4Re is used to denote a multitude of things. For once it is a set of libraries that abstract from the bare Fiasco.OC interface. It also adds user-level defined protocols that help in the construction of dynamic systems. Some of these protocols are used by servers to provide services to other system components. A *server* is one task with one or more threads executing in it and usually provides a service to other system components. L4Re provides the servers moe (the resource manager), sigma0 (the root-pager), ned (the launch control agent), io (the device manager) and many other servers with device

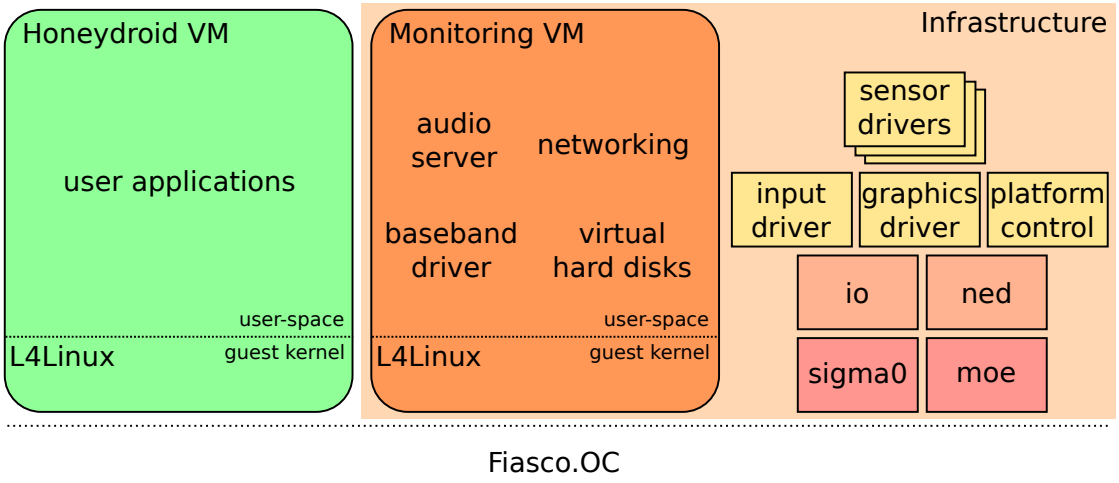

Figure 5.1: Honeydroid: Architectural overview

driver functionality with a varying degree of hardware dependency. In Figure 5.1 we see the collection of servers that provide basic services to the system on the right.

5.1.2 Compartment

A loosely coupled collection of tasks, with one or many threads in VCPU mode executing inside them, we refer to as a *compartment*. The kernel-task of L4Linux together with the secondary tasks for its processes form a compartment. In Figure 5.1 there are two such compartments. The Monitoring compartment in the middle and the Honeydroid compartment on the left. The Honeydroid compartment runs all the services of the Android middleware as well as Android applications. Thereby providing a fully functional Android experience to the user. The Monitoring compartment, like the servers discussed before, belong to the infrastructure. It provides services to the Honeydroid compartment. Using a L4Linux compartment to provide services has several reasons. For once it is convenient to reuse infrastructure that has no native L4Re counterpart yet and is to complex for a quick implementation (e.g. an IP-network stack). Also, as this is a scientific research vehicle we have to deal with of the shelf hardware with little vendor support for development. Therefore some services can only be provided by reusing binary-only libraries which are binary compatible to Linux but not to Fiasco.OC/L4Re.

5.2 Platform Control

A considerable part of the infrastructure is concerned with controlling low level functionality of the platform. The following services are provided by servers.

- **Power** The power service controls the power management IC (PMIC). The PMIC controls the power supplies for various peripherals as well as the SOC and certain individual building blocks of the SOC. Peripherals that need proper power supply configuration are:
  - The baseband (3G modem),
  - the MMC flash storage,
  - the GPU
- **Backlight** The backlight service controls the brightness of the screen as well as the power state of the display device.
- **Clock** The clock service controls the gating of the clocks for various cores within the SOC (e.g. CPU, GPU, peripheral bus controllers, timers ...).
- **Battery** The battery service provides the honeydroid compartment with the charging and fuel state of the battery.
- **Charger** The battery charger driver is also part of the system control infrastructure.
- **Vibrator** The vibrator motor driver service allows the honeydroid compartment to give tactile feedback. It draws on the capabilities of the power service to provide this service.
- **RTC** The real time clock service provides the honeydroid compartment with wall clock time.
- **Jack** The jack service provides information about whether or not headphones are attached to the devices head phone jack.

5.3 Networking

The L4 Shared Memory Network driver (l4shmnet) is L4Linux NIC driver that uses shared memory and asynchronous signaling as a back-end and connects two L4Linux

instances. Thus Internet protocol (IP) links can be established between instances of L4Linux. One such link is established between the monitoring compartment and the honeydroid compartment. This link is used to provide mobile data access as well as additional services such as block device support or audio.

5.3.1 Modem

The control channel of the android radio interface layer (RIL) is virtualized between the Java RIL layer and the native code RIL daemon (RILD). All communication between those two is done via an Unix domain socket (/dev/socket/rild). For virtualization, the RILD runs in the monitoring compartment, and forward the socket communications via the l4shmnet link to the honeydroid compartment. There is no need to modify RILD or RIL. Socket forwarding is done with a port of the open source socat tool. The data channel in Android works like this:

1. RIL sends a command to modem to establish a data context (pdp context).
2. The modem establishes a connection and reports success to RIL.
3. Upon connection establishment the Linux modem driver creates a network device (e.g. `rmnet0`).
4. All mobile data is then sent and received via this network device.

Virtualization of this data channel is done with network address translation (NAT). The monitoring compartment acts as router between the local network, which is set up between the monitoring compartment and the honeydroid compartment, and the Internet reachable via `rmnet0`. The monitoring compartment sets the required iptables rules to do NAT. The honeydroid compartment, in turn, sets the IP address of the monitoring compartment as its default gateway. Once the monitoring compartment has a network connection, all network traffic from the honeydroid compartment will be able to make use of this connection.

5.4 Input

A dedicated input driver server drives the touch screen and GPIO connected hardware buttons. It reports input events directly to the honeydroid compartment. Input events are delivered via the L4Re::Event protocol. The L4Re::Event protocol reports new events through asynchronous notification with a payload placed in a shared memory buffer.

L4Linux has an event driver, that provides a standard event interface to its user-space and draws on the L4Re::Event protocol as its back-end.

5.5 Output

5.5.1 Graphics Stack

The graphics stack comprises a display-controller-server and GPU-server. The display-controller-server provides a frame-buffer to the honeydroid compartment by means of the *L4Re::Goos* protocol. L4Linux has the L4-FB driver. This driver provides default frame-buffer functionality to the user-space of L4Linux while using the L4Re::Goos protocol as back-end. For efficient rendering the honeydroid compartment can draw on the capabilities of a graphics-processing-unit (GPU). GPUs, however, are very powerful and freely programmable DMA devices which, provided one has enough control over it, can be used to evade memory isolation imposed by the operating system. To restrict the memory access capabilities of the GPU, access to the GPU is mediated by a back-end virtualization scheme. In order to allow for optimal performance the GPU-server, that mediates GPU access, can configure the GPU in such a way that it renders directly to the frame-buffer provided by the display controller driver.

L4Re::Goos protocol

L4Re introduces the **L4Re::Goos** protocol. The goos protocol knows the primitives *screen*, *view* and *buffer* (see figure 5.3). Screens denote the display area of a physical display device. Buffers are memory regions meant for the storage of digital representations of images. Via the goos protocol buffers or sections thereof can be attached to views which in turn have a size and a position on a screen. A subset of the goos protocol can be used to provide simple frame-buffer functionality. For example, a view that spans the whole screen is backed by a buffer that represents the content of this whole screen view.

5.5.2 Audio subsystem

A custom audio-server runs in the monitoring compartment. This audio-server receives the output audio data from the honeypot compartment, mixes it if necessary and plays it back on the native hardware. Furthermore, the audio-server can send the microphone/input data it receives from the native audio hardware back to the honeypot compartments. To access the hardware, the audio-server uses the native libraries from Samsung Galaxy

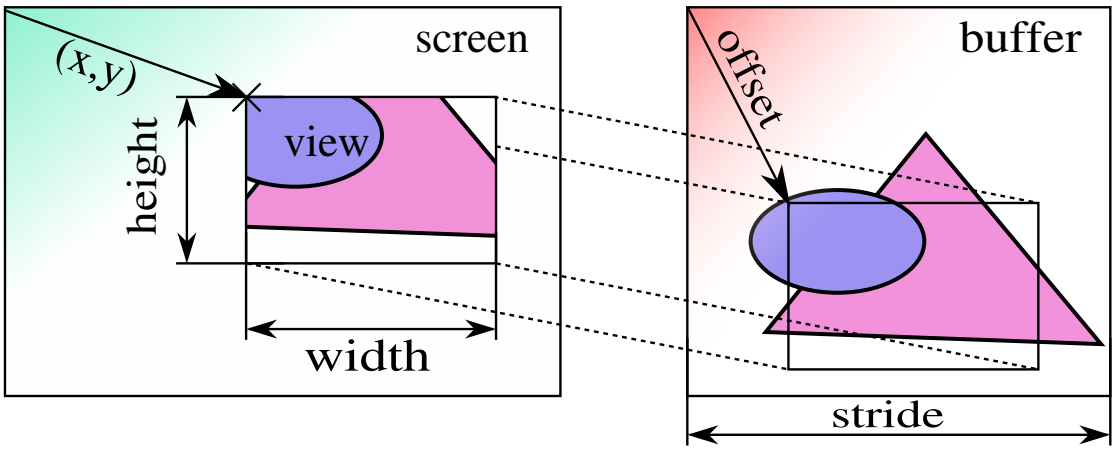

Figure 5.2: Goos protocol primitives.

S2. To do so, unchanged versions of the Android AudioFlinger and the AudioPolicyService run in the context of the audio-server. The AudioFlinger is also used to mix the data from the client compartment (honeypot compartment). In addition to the vanilla Android audio services, a Yamaha media service runs in the audio-server, which is required by the native Galaxy S2 audio library. The Yamaha media service is accessed via a shared library.

A complete media-server runs in the client compartment with a stub library. The stub library's interfaces are implemented in a way that they send/receive the audio data to/from the audio-server in the monitoring compartment.

5.6 Mass Storage

The monitoring compartment is responsible for providing mass storage services to the honeypot compartment. For this purpose, a Network Block Device [4] (nbd) server instance listens on multiple ports, each granting access to a different storage back-end. These back-ends do not need to be physical media but can also be represented by virtual devices, such as Logical Volume Manager (LVM) volumes or loop-back devices. The choice of storage medium is completely transparent to the honeypot compartment, which uses nbd-client to connect to the nbd-server process running on the monitoring compartment using a l4shmnet link. The initial setup takes place during boot-up to ensure the system volume's availability. Storage devices are set up as nbd block devices

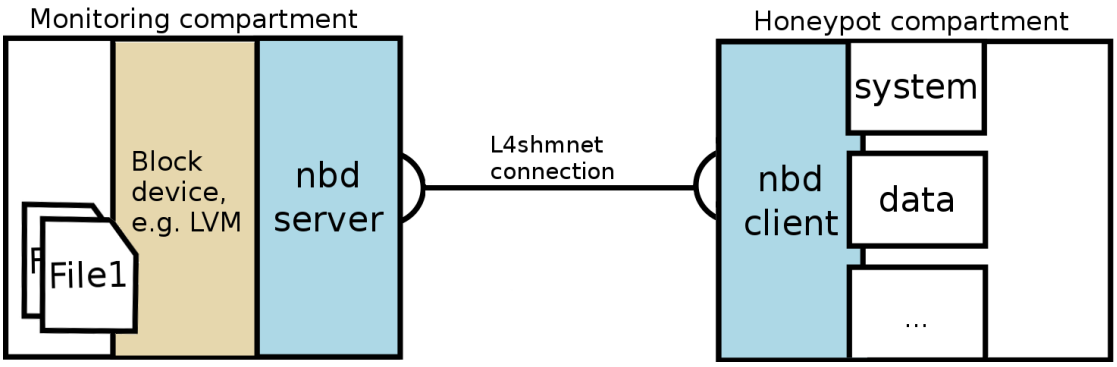

Figure 5.3: nbd setup

by the nbd-client kernel module, and can be mounted like any regular block device. On the server side, nbd runs as a regular process in L4Linux user space.

5.7 Anomaly Sensors

One goal of the mobile honeypot is to have sensors outside the reach of malware. The assumption is that the attacking malware may be capable of subverting the guest kernel but can not break out of its virtual machine. The lightweight malware detector (LMD) described in [28] requires the frequency of a distinct set of system calls issued within the honeydroid compartment as well as the state of the device's screen.

5.7.1 System call sensor

In Section 2.5 we covered the primitives of Fiasco.OC and the VCPU feature. In Section 2.6 we also gave a brief introduction to L4Linux. For the system call sensor the way system calls are handled by L4Linux is exploited. Figure 5.4 shows that when a user-process on L4Linux issues a system call control is passed ① to the underlying μ-kernel, Fiasco.OC. The up-call mechanism of Fiascos.OC's VCPU then injects ④ the system call into the guest kernel. In the mobile honeypot this mechanism is intercepted ② in the μ-kernel and the system calls are counted ③ in a new kernel object called *Systrace*. Systrace is a new singleton kernel object that can be referred to through a capability. It exports a unique interface to allow for querying the system call counters by a user-space agent. The monitoring compartment is entrusted with the capability to the new object. It features a driver ⑥, which provides the service to its user-space. The LMD can now

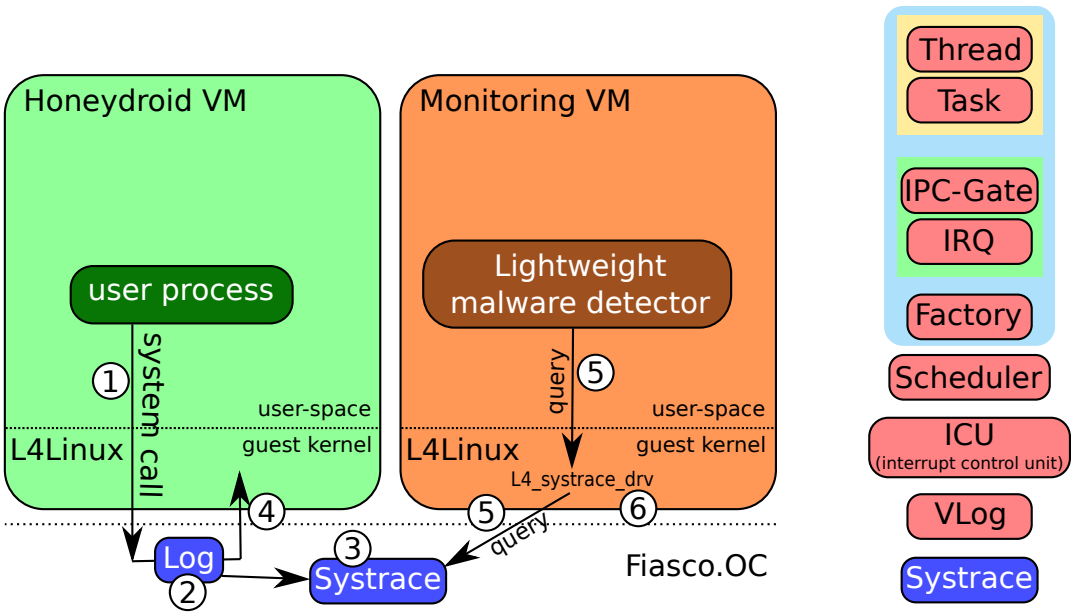

Figure 5.4:

draw on this service to periodically query ⑤ the counter state.

The monitoring compartment being an instance of L4Linux itself also issues system calls, which go through the same VCPU mechanism. Systrace, however, ignores system calls originating from the particular compartment, that also issues queries. So it must be mentioned that the current implementation only works if there are two compartments in the system of which one does the monitoring. Otherwise the counted values are a composition of two or more system call event sources.

5.7.2 Screen state sensor

The state of the screen is available in the platform control server of the mobile honeypot infrastructure (see Section 5.2). A new communication channel was introduced for the monitoring compartment to query the screen state. L4Linux features a new driver to provide an this service to its user-space. The LMD can draw on this interface to poll the screen state.

5.8 Forensic Data Acquisition

For the purpose of forensics, it is desirable to have a snapshot of the honeypot compartment's mass storage from the time an anomaly was detected. In addition to that, to simplify the process of reverting to a clean system after the analysis is finished, a snapshot of the initial state could be used. To prevent malware from affecting the snapshotting process, it must be handled by a separate compartment. Since the monitoring compartment is already in charge of mass storage, we place a snapshotting mechanism between nbd-server and its back-end. LVM, a popular software for managing logical volumes that relies on the device mapper interface provided by the Linux kernel, offers this functionality. As we already mentioned in Section 5.6, it seamlessly integrates with nbd. We can use flat files or regular block devices such as partitions on the MMC card to create physical LVM volumes. These volumes can then be grouped into volume groups, which can be divided into logical volumes. Logical volumes are available to the system as block devices and can for example be used to house file systems. We create a data and a system logical volume and configure nbd-server to make them accessible to the honeypot compartment. Upon detection of an anomaly, a copy-on-write snapshot is created for both volumes, after which all further writes to the respective volume cause the affected block to be copied to the snapshot volume before modification. The snapshot volume represents the state of its origin at the time the snapshot was made and can later be used for forensic analysis. It is a also possible to use snapshots to revert its origin to a previous state, which fulfills our secondary requirement.

5.9 Premium SMS Filter

As listed in Section 4.3 of Chapter 4, containment of malware that is capable of sending premium SMSs is paramount. This section describes the design and implementation of a premium SMS filter in honeydroid.

Section 5.3.1 describes the overall design and implementation of the modem driver in Honeydroid. As noted in the section, the monitoring compartment servers as a proxy between the Honeypot VM and the telephony infrastructure i.e., base stations and core network. This permits us to implement a premium SMS filter in the monitoring VM.

5.9.1 SMS Routing

In vanilla Android smartphones, SMS payload is encoded by Android's Radio Interface Layer Daemon (RILD) that is packaged as part of Android's system messaging application. The encoded payload consists of meta data about the SMS, such as, the SMS

Center (SMSC) number, the destination telephone number etc., apart from the actual content of the SMS. The RILD forwards the payload to a Unix socket (/dev/socket/rild) that interfaces with the vendor modem library[1]. Finally, the modem library transmits the encoded SMS payload to the nearest base station and is eventually delivered to the recipient.

On Honeydroid, SMS routing involves inter-VM communication with the monitoring compartment acting as a proxy. Virtualization based code isolation dictates that vendor's modem driver reside in the monitoring VM. Since the messaging application and Android's RILD reside in the Honeypot VM, sending an SMS involves multiple hops. First, the messaging app relays the encoded SMS payload to the rild Unix socket in the Honeypot VM; then, the `socat`[2] program forwards the payload to it's counterpart in the monitoring VM over l4shmnet—L4Linux's shared memory networking channel; finally, an instance of socat in the monitoring VM forwards the payload to the vendor's modem driver in the same VM.

Since l4shmnet is a network-based inter-VM communication channel, transmission of the SMS payload from the Honeypot VM to the monitoring VM takes place over a networking protocol—in our case it is the User Datagram Protocol (UDP). We exploit this implementation quirk and piggyback on iptables[3] to filter out undesirable premium SMS messages.

5.9.2 Content Filtering

Content filtering on iptables involves setting up filtering rule sets involving hex string signatures that match the expected payload. In the context of SMS filtering, this involves reverse engineering the payload structure i.e., it's header fields and offset to obtain the destination telephone number.

Having obtained this information, it is simple to craft rules such that UDP packets that encapsulate SMS payloads to blacklisted telephone numbers are silently dropped in the monitoring VM proxy.

Example Scenario Rules to block premium SMS numbers on the device can be added as iptables rules on a per number basis. Let us suppose you want to block outgoing SMSs to the number '82555'. You can do so as follows:

- The template rule for blocking outgoing SMSs is

[1]The vendor library for Samsung Galaxy S2 is proprietary.
[2]Socat is a popular networking utility program that serves as a proxy for data packets
[3]iptables is a network firewall utility

```
iptables -I SMS_FILTER 1 -p udp --dport 2342 -m string --from 68 --to
    to_bytenumber --hex-string "|RILD encoded destination number|" --algo kmp -j
    DROP
```

- Two fields that are user supplied are the `to_bytenumber` and `RILD encoded destination number`
- `to_bytenumber` can be calculated as follows

```
let N = number of digits in the destination number
if N is odd
to_bytenumber = 68 + 2 * (N + 1)
else
to_bytenumber = 68 + 2 * N
```

  For the number ‘82555‘, this number is 80 i.e., 68 + 2 * (5 + 1)

- The `RILD encoded destination number` is essentially a wrapper encoding over SMS PDU standard. To compute the encoding for the number you want blacklisted, you need to stick to the following rules:
  - Before encoding, split the destination number into groups of 2 digits, padding the lone digit at the end (if N is odd) by ‘f‘ e.g., 82555 becomes ‘82‘, ‘55‘, and ‘5f‘
  - Swap the digits in each group e.g., ‘82‘, ‘55‘, ‘5f‘ become ‘28‘, ‘55‘, ‘f5‘
  - Then, encode each digit in the above sequence leftmost to rightmost in 16 bit/ 2 octet format. Let’s suppose we want to encode the number ‘2‘
  - The most significant octet is the ASCII representation of the digit itself. For ‘2‘, this is the hex byte ‘32‘
  - The least significant octet is always zero i.e., hex byte ‘00‘
  - The rild encoded sequence for ‘82555‘, is ‘32003800350035006600350 0‘
  - Finally, our iptables rule to blacklist the number ‘82555‘ becomes

```
iptables -I SMS_FILTER 1 -p udp --dport 2342 -m string --from 68 --to 80 --hex-string
    "|32003800350035006600350 0|" --algo kmp -j DROP
```

6 Communication with Service Provider

6.1 Introduction

This chapter describes the specification of the communication between a mobile anomaly detection (AD) system running on a Honeydroid device to a centralized, operator-hosted data collector. The system relies on the following facts:

1. The mobile AD Honeydroid system is Android-based or has the same capabilities of it
2. The AD developer and the data collector party that hosts the server are able to communicate out-of-band to share a common secret in a secure way.

The system is based on the following protocols/standards:

1. HTTPS: to secure the channel used to transmit data
2. AES: to protect the pre-shared secret
3. JSON: to exchange data.

6.1.1 Anomaly detection message from the mobile Honeydroid to the collector

The AD mobile nodes use a simple way to communicate data to the data collector. They open a HTTPS connection to the collector (its public certificate must be previously exchanged between parties), and use the HTTP header to send the collector the pre-shared key, hashed with sha256 and in base64. This key will be used to authenticate the node. The authentication part of the header will look like:

```
Authorization: Token token="9e88bdb88f04e5aba19f47772830501b9320bf669eb
25e5399959a9aa20709b9
```

The body of the message will be a POST containing a JSON object (collection of key/values) with the information to send to the collector. The JSON message will contain:

- node: an identification id for the node (can be the ANDROID_ID, or the IMSI of the sim card or the phone number of the user, or any other string used to identify the node)
- owner: an identification string used to identify the owner of the node
- timestamp: local timestamp of the node, in Unix timestamp form
- aid: identification of the anomaly
- ainfo [opt]: an optional JSON that contains additional info (such as md5/sha of binaries, IP addresses) depending of the anomaly itself.

The suburi used to send data to the collector will be `/adsend`.

Here follows an example of a message coming from a node, for the anomaly labeled a1, giving that the collector is hosted on the server collector.be-secure.it:

```
POST /adsend/ HTTP/1.1
Host: collector.be-secure.it
Authorization: Token token=9e88bdb88f04e5aba19f47772830501b9320bf669eb
25e5399959a9aa20709b9

{
  "node": "123456789",
  "owner": "partner1",
  "timestamp": "1401098489",
  "aid": "a1",
  "ainfo": { "appmd5": "b8fc7876e5e69f356837d87de8a1427e" .}
}
```

6.1.2 Response message from the collector to the mobile Honeydroid

Once the collector has processed the message (i.e., verified the authentication, processed the body, checked its validity and stored it in the database), it returns back a message containing the result of the operation. The response contains a HTTP code and a JSON, which shows the details of the result (this is necessary in case an error occurred). The HTTP response code will be:

- 200 - OK: the message has been correctly processed
- 422 - Unprocessable Entity: the message could not be processed.

The latter will be followed by a JSON containing the key 'error' and the details of the error, such as, for example:

```
HTTP/1.1 422 Unprocessable Entity
Content-Length: 41

{
  "error": "Field node must be present."
}
```

In this example the node can understand that its message has not been correctly processed because it did not provide the node field in the JSON.

7 Evaluation

7.1 Classification of Performance and Benchmarks

Identifying useful performance metrics is of importance in comparing multiple systems. The metrics should allow the user to easily identify key bottlenecks in the system. It must also account for overheads and latencies of different components of a system. Since there are performance issues at a micro and macro level a combination of such metrics can provide information on the behavior of the system. Micro-level metrics can be useful to compare subsystem performances but that may not translate into macro-level results i.e the user may not be affected by bad micro-level performance. While macro-level metrics can provide information about the whole system or a whole component, it may not provide enough information to identify the real performance inhibitor like micro-level benchmarks.

7.1.1 Classification of Performance

To measure the performance of the smartphone from a micro-level up to the macro-level we decided to classify our performance metrics as throughput, latency or response time and power. The throughput can be in terms of Million Floating-point Operations(MFLOPs), Million Instructions per Second(MIPs), Kilobits per second(Kbps) or Megabytes per second(MBps). The latency is in the order of seconds(s). And finally the power is in terms of Watt(W).

7.1.2 Classification of Benchmarks

In [35], the authors classify benchmarks into four categories:toy programs, synthetic, kernel and real. A toy benchmark is one that runs a program such as the Tower of Hanoi. Synthetic benchmarks perform an average number of operations that is neither too high nor too low, popular examples are Dhrystone and Whetstone.. A kernel is a benchmark that uses say, the main loop of a scientific program, Linpack or Scimark2 are prime examples of that. A real benchmark is one that runs a real program like gcc for instance. SPEC(Standard Performance Evaluation Corporation) [22] is the most notable consortium that benchmarks using real programs.

The above classification of benchmarks is not representative of our work. We wanted to identify categories to be general enough so that the classification can be applied to any smartphone or portable mobile device. Therefore, we classified benchmarks as Networking, Operating System, CPU, Application, and Power. Under each of these categories, the benchmark can be a toy, synthetic, kernel or real benchmark. This classification enables us to benchmark the smartphone in a modular way, it also allows us to the view and analyze the results from the benchmarks easily. And lastly, by combining the benchmarks we obtain an aggregated view of the smartphone.

Networking

In todays realm of computing, networking is central to communication between devices. The network performance plays a major factor in data transfer such as web browsing and multimedia applications such as skype and live streaming. The network performance of smartphones depend on the modem used such as GPRS, 3G, 4G or Wifi as well as the various protocols implemented in the operating system. TCP and UDP are key Transport protocols that can be used to measure the performance in terms of throughput and latency. Another governing factor of performance can be the Maximum Transmission Unit(MTU). Since there are so many factors here benchmarking the networking by itself away the Operating System is a good idea. Iperf is a synthetic benchmark that can be used to evaluate network performance.

Operating System

This category deals with benchmarking the performance of the Operating System primitives such as system calls, memory access, memory write and job scheduling. The list can be expanded to focus on specific parts of the operating system such as interrupt handling, filesystem management and so on. We limited our work to system calls, the memory subsystem and scheduling. The performance of the OS kernel and user-land processes is key to low latency and high throughput. The benchmarks that are used here are mainly synthetic and real benchmarks. The system call benchmark programs are real while the memory subsystem and scheduling benchmark programs are synthetic.

CPU

Performance of the CPU is key to any computer, it needs low latency and high throughput. There is a long history of benchmarking CPUs and it was the reason that so many types of benchmarks and benchmark categories were created. But what has been understood is that the perfect benchmark is a real program, but that is hardly used. The

performance of the CPU can be benchmarked using kernel benchmarks like Linpack, Scimark2 and RealPi. Linpack and Scimark2 measure the throughput while RealPi measures the response time. Although it is well known in the computer architecture and systems community that measuring the throughput of a CPU in terms of Million Floating-point Operations per second or Million Instructions per second is non-indicative of true CPU performance [29], it may be used to compare different Operating Systems on the same hardware. That is why we have included such benchmarks as part of our evaluation.

Application

In general, applications are programs that run in the user-land address space of the OS. These programs are the interface between the user and the OS. Measuring the performance at the Application layer provides us with high-level data points that can be used to compare against micro-level benchmarks such as system-call benchmarks. An example application is the user-land of the Android OS such as the Dalvik VM or the Javascript engine of the web browser. These are programs that are exposed to the user, and therefore the response time matters in certain cases such as touch screen events while the throughput matters in cases such as handling multiple Javascript events at once. Our application benchmarks are mainly toy benchmarks such as Sunspider for the Javascript engine and synthetic benchmarks like GCBench for Dalvik VM benchmarking.

Power

Smartphones are completely dependent on batteries for their energy source, therefore, the energy consumed by the hardware and software affects the life of the battery. It is why power management is very critical in modern day smartphones. Since a phone can be in so many different states(idle, screen on, 3G surfing, wifi on, speakers on) it is difficult to have uniform measurements, but comparisons between phones can be made in particular states such as the idle state-when the radio is on but screen is off or while a voice phone call is in progress, here again the screen is off but the radio, speaker and microphone are active. Such measurements can be used to compare battery performance. These benchmarks can be considered real benchmarks and measured in Watt(W).

7.2 Methodology

One of the main goals of the work conducted was to obtain accurate information that can be used to compare the performance of Honeydroid to that of an unmodified Android

hereafter referred to as Vanilla. In order to obtain data for comparison a number of benchmarks were executed on Vanilla and Honeydroid in a systematic and repeatable manner using the proposed evaluation framework. The resulting information collected was then converted to a ratio form similar to the SPECRatio. The ratio represented the speedup of Honeydroid to Vanilla. The speedup from each benchmark was then plotted as a graph. For the network tests, total network throughput was plotted instead of ratios as the throughput for GPRS was comparable to Vanilla. The same was applied towards the power test plots.

7.3 Evaluation Framework

A graphical representation of our evaluation framework is shown in 7.1. We will refer to the benchmarks as Tests over here. It consists of: 1) Device Under Test(DUT) 2)Test Monitor 3)Test Result Parser and 4)Data Visualizer. The DUT is where the different tests are executed. It is connected to the Test Monitor via a management interface(USB) which is the main communication channel between the DUT and the Test Monitor. The Test Monitor executes Test cases which are either automated scripts or manual tests due to hardware or test case constraints on the DUT. Once the DUT is tested the results are collected by the Test Monitor and passed on to the Test Result Parser which transforms the results into data that can be used by the Data Visualizer for visualizing the results.

7.4 Experimental Setup

The smartphone used for the experiments was a Samsung Galaxy S II - Model GT-I9100. The system configuration of the phone is

- CPU - 1.2GHz dual-core ARM Cortex-A9
- GPU - ARM Mali-400 MP4
- 1 GB RAM
- Super AMOLED Plus(480x800) pixels and RBG-Matrix
- SoC - Samsung Exynos 4 Dual 45nm
- 3G UMTS

The phone(DUT), was connected to a PC(Test Monitor) running Ubuntu 12.04 via a Samsung Anyway S101. 7.2 depicts the setup. Based on the configuration on the Anyway

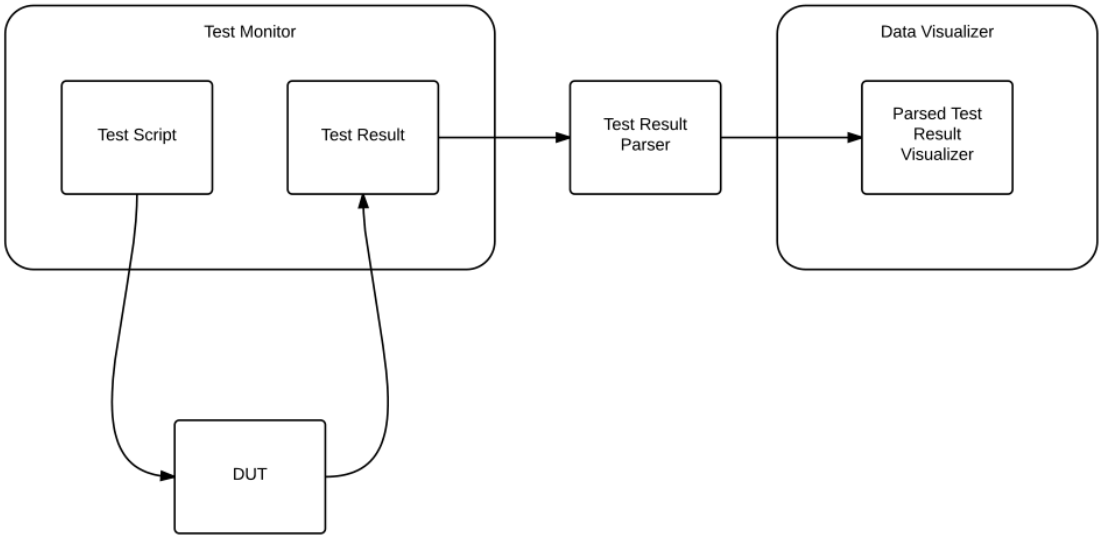

Figure 7.1: A block diagram of the performance evaluation framework

S101 the PC connects to the DUTs USB or UART, it cannot be connected to both at once. The Anyway S101 has current inputs of 2 types, one for powering the itself and the other to provide the DUT with current. A DC Power Supply Agilent E3645A was used to provide the Anyway S101 with current for the phone at a Voltage of 4.3V.

To emulate a Mobile Base station, a USRP-1 [16] device was setup which can be seen on the left of the photograph. The DUT had a SIM card that was configured to connect to the setup Mobile Base station. OpenBTS 4.0 and GNUradio 3.4.2[17] were software's used to control the USRP device and host a GSM and GPRS station. OpenBTS was configured to have most bandwidth and down-link capacity by setting Channels.Min.C0=6. To measure the energy consumption, an Agilent U1253B Ammeter was connected in series with the Anyway S101 box and DC power supply along the positive line. These devices can be seen in the upper middle and upper right of the photograph. The Ammeter was able to log the current every second. For measuring the energy consumption for a voice call, a commercial SIM card was used that connected to Deutsche Telekom using a GSM connection.

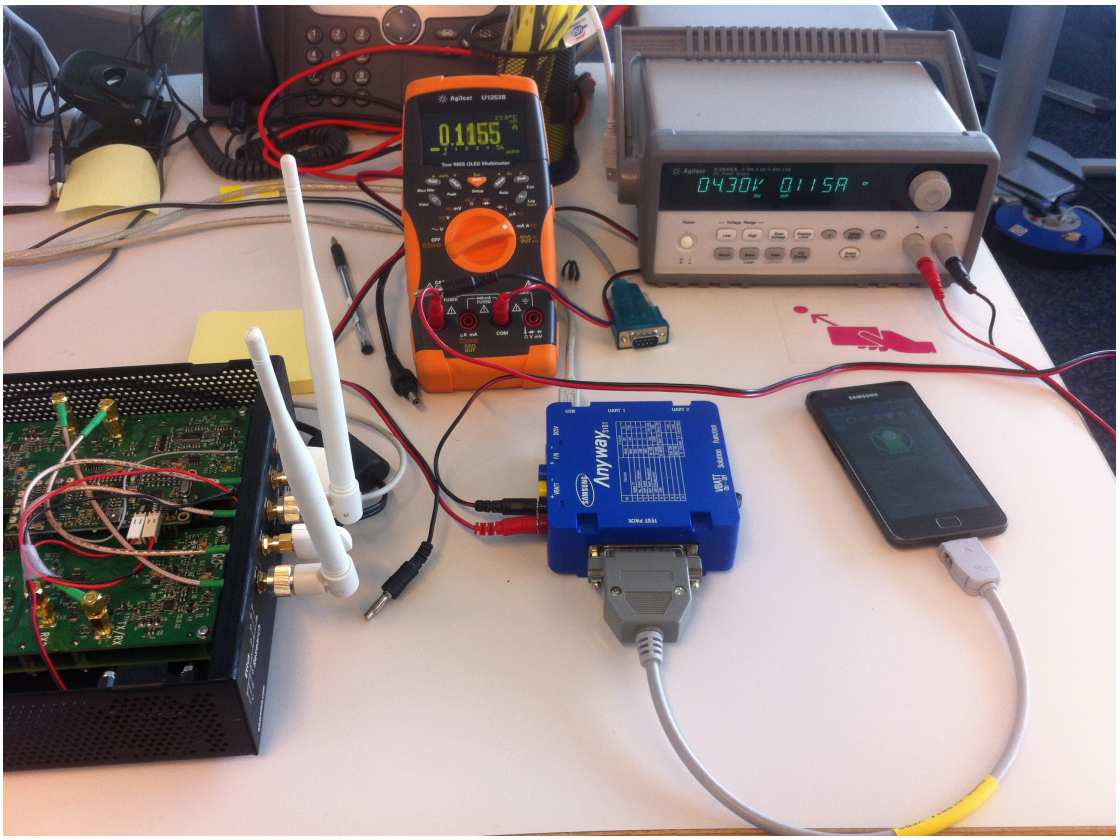

Figure 7.2: A photograph of the experimental setup

7.5 Network Evaluation

The network tests were conducted over the GPRS data service. Although, GPRS itself is a bottleneck with respect to data rates, it does shed light on the performance difference between Honeydroid and Android. The network experiments conducted were broadly divided into TCP and UDP tests. The TCP tests were further classified into Up-link and Down-link tests. Each Up-link and Down-link test was conducted with an MSS=536,1500 and varying total data transfer amounts from 128KB to 256KB to 512KB to 1024KB to 2172KB. The TCP and UDP server/client programs were facilitated by Iperf 2 which ran on the PC and smartphone. The Iperf benchmark is a real benchmark.

7.5.1 Iperf 2

Iperf 2[20] has two modes of operation, server and client mode. It allows the user to configure the ports, IP address, data transfer size, Maximum Segment Size(MSS), Maximum Transmission Unit(MTU), etc. An Android version of Iperf was obtained from [19] which had an iperf ARM executable binary packaged within it. The iperf binary was executed through a command line shell instead of incurring the overhead of the app. The behavior of Iperf in TCP mode is as follows:

- TCP server listens on a random or specified port for an incoming TCP connection.
- TCP client binds to a random or specified port and then establishes a TCP connection with the Iperf TCP server on the specified port.
- The client then transfers the specified amount of data through multiple packet transfers, while the server acknowledges the data.
- Once the data transfer is complete the connections are terminated.
- A report of the TCP throughput along with the time taken and total data transferred is displayed to stdout.

The behavior of Iperf in UDP mode is as follows:

- UDP server listens on a random or specified port for incoming UDP datagrams.
- UDP client binds to a random or specified port and then sends out UDP datagrams destined to the Iperf UDP server.
- The client then transfers the specified amount of data through multiple datagrams.

- The server receives the datagrams.
- A report of the UDP throughput along with the time taken, link jitter, and packet loss is displayed to stdout.

Down-link Tests

The down-link tests measure the down-link network capacity of the smartphone. The down-link capacity reported by Iperf is the mean value of the throughput measured every second. For these tests, Iperf runs as the server on the smartphone while the PC has Iperf running as the client.

Up-link Tests

The up-link tests measure the up-link network capacity of the smartphone. The up-link capacity reported by Iperf is the mean value of the throughput measured every second. For these tests, Iperf runs as the client on the smartphone while the PC has Iperf running as the server.

MSS and MTU

The MSS is the size of the Transport layer packet. The smallest size for TCP allowed by Iperf is 536 bytes. This size include the header and payload fields which are encapsulated into the IP packet. The MTU is the size of the Data later frame. This is dependent on the Layer 2 technology such as Ethernet, Frame Relay, etc. In most practical cases the MTU of an Ethernet v2 frame is 1500 bytes. RFC 1191[42] describes Path MTU Discovery as a technique for determining the lowest MTU between the source and destination. This size affects the size of the above layer payloads. Therefore, conducting tests with different MTU sizes can expose performance issues with the network stack implemented in Honeydroid.

Data size transfer

The total amount of data transferred between the client and server was varied from 128 Kilobytes to 2172 Kilobytes. Short data transfers may incur more overhead than long data transfers, also since Iperf averages the total throughput of the network test, transferring a large amount of data would provide a better averaged number than a shorter data transfer. It would also allow us to see if there is an actual difference between short and long data transfer throughput and data loss.

7.5.2 Honeydroid and Vanilla Down-link Tests

To measure Honeydroid's down-link TCP throughput, three extra IPtables[18] rules had to be inserted. These rules were inserted into the Infrastructure compartment of Honeydroid. The rules inserted were

- adb -s 53694D4B6F33 shell iptables -I FORWARD 1 -i pdp0 -o eth0 -m state –state NEW,RELATED,ESTABLISHED -j ACCEPT
- adb -s 53694D4B6F33 shell iptables -t nat -A PREROUTING -p tcp -d 172.16.18.100 –dport 5000 -j DNAT –to-destination 169.254.2.2
- adb -s 53694D4B6F33 shell iptables -t nat -A PREROUTING -p udp -d 172.16.18.100 –dport 6000 -j DNAT –to-destination 169.254.2.2

The above rules were necessary to allow incoming TCP and UDP connections to be forwarded to the Honeydroid compartment which is where Iperf runs. These rules are not allowed otherwise, as it would be a security hole in the phone, allowing attackers to connect to the phone from the Internet. No such Iptables rule was necessary for the Android OS.

pdp0 is the interface with GPRS/Internet connectivity on Honeydroid and eth0 is a virtual Ethernet interface used for inter/intra-compartment communication.

The first rule allows all incoming traffic on pdp0 thats destined to eth0 to be forwarded if the connection is new(like the Iperf incoming connection), related or established(such as connections from web-servers that were initiated by Honeydroid).

The second rule explicitly forwards and translate all TCP traffic destined to pdp0 IP address 172.16.18.100 on port 5000 to the Honeydroids compartment IP address 169.254.2.2

The third rule explicitly forwards and translate all UDP traffic destined to pdp0 IP address 172.16.18.100 on port 6000 to the Honeydroids compartment IP address 169.254.2.2

A shell script was written to automate the execution of the tests and collection of test results.

The script runs the iperf client and server on the respective hosts. The iperf server reports are stored in a server file while the client iperf reports are stored in a client file. After all tests are executed, the client and server files are parsed to obtain throughput results which are then displayed to stdout and also saved to a results file which are then used by the data visualizer to plot the graphs.

7.6 Operating System Evaluation

In this section we evaluate the OS in three different ways, the system call latency, the system call memory bandwidth and the kernel scheduler. A more detailed description follows starting with the system calls using Lmbench and then moving towards the kernels job scheduler using Hackbench.

Measuring the time taken from when a system call is invoked to the time the result is returned is considered to be the system call latency. The bandwidth of a system call is the amount of data it can move across memory in a given time. Evaluating the performance of system calls allows us to notice the latency and bandwidth of the underlying subsystems. The latencies and bandwidth of system calls were measured using a well developed benchmarking tool called LMbench[cite website]. The LMbench benchmark program is a combination of synthetic and real benchmarks as certain tests are real such as the getppid test while certain tests like the read are synthetic.

7.6.1 LMbench

Lmbench[7] is a micro-benchmarking suite that was developed by Larry McVoy and Carl Staelin. The suite encompass tests that measure memory bandwidth, IPC bandwidth, Cached I/O bandwidth, Operating system entry/exit, Signal handling, Process creation, Context switching and File system latency. Lmbench has become a very popular and trusted benchmarking suite used by performance evaluation teams and system designers[41],[26] to identify system level bottlenecks. Unlike old versions of SPEC that specifies performance in terms of MFLOPs or MIPs, LMbench focuses on providing the user with low level system(kernel) performances metrics such as latency or bandwidth which can be used to improve or uncover critical performance issues that may surface at the user level.

The Lmbench source code written in C, was included in the Honeydroid build system so that the final raw image would have the LMbench binaries present in the filesystem. The obtained binaries were used for Vanilla as well. They were transferred to the device by using the 'adb push' command.

- Latency tests
- Bandwidth tests

Latency tests

Since there are different categories of system calls and LMbench can test many parts of the kernel we decided to focus on only a few kernel operations where latency is a key

issue. We identified these areas to be kernel entry/exit, page faults, memory/file system, process creation/destruction and signal handling. IPC tests were not covered as there has been extensive research done in the area[39], [33], [27]. Lmbench has tests that cover the mentioned areas of interest through the following system calls:

- **getppid:** This test program measures the time to get the parent process id. The system call getppid() does very little work inside the kernel and it is therefore a good test to gauge the kernel entry and exit time.
- **read:** This test program measures the time to read one byte from /dev/zero. This is a system call that is used very often by Linux. Therefore, measuring the latency of the read operation is important in comparing Honeydroid with Vanilla. It reads up to n bytes from a given files file descriptor. In this case it reads 1 byte from the /dev/zero file descriptor.
- **write:** This test program measures the time to write one byte to /dev/zero. The write() operation is another File management system call that is frequently used by Linux. It writes up to n bytes from the buffer to a file specified by a given file descriptor. In this case it writes 1 byte to the /dev/zero file descriptor.
- **stat:** This test program measures the time to stat() a file whose inode is already cached. stat() is used by applications such as ls in Linux to obtain file attributes about an index node(inode) which is a filesystem object. This is another system call that is used frequently.
- **fstat:** This test program measures the time to fstat() an open file whose inode is already cached. fstat() is similar to stat() but instead, the file that requires a stat() operation is specified by its file descriptor.
- **open:** This test program measures the time to open() and close() a file. The open() and close() operations are very frequently used system calls. It returns a file descriptor to a given pathname which corresponds to an entry in a system-wide table of open files. The close() operation closes a given file descriptor. This makes sure that the file descriptor resources are freed and may be used again.
- **pagefault:** This test program measures the time a page of a specified file can be faulted in. The file is flushed from (local) memory by using the msync() interface with the invalidate flag set. The entire file is mapped in, and accesses the pages backwards using a stride of 256K Kilobytes. Page faults are expensive operations as they involve CPU interrupts and privilege switches which involves register push

operations and page table and cache flushes. The exception handling and signaling involved are other contributors to its high cost of operation.

- **fork:** This test program measures the time to split a process into two(nearly) identical copies and have one exit. This is how new processes are created but is not very useful since both processes are doing the same thing. The focus of this test is to measure the time take to just create a new child process and then exit right after that.
- **exec:** This test program measures the time to create a new process and have that new process run a new program. This is the inner loop of all shells. This test is similar to the previous test but only differs in its behavior after creating the new child process. Here a new program is run by the created child process
- **shell:** This test program measures the time to create a new process and have that new process run a new program by asking the system shell to find that program and run it. It's a very expensive operation and most realistic in its behavior. Therefore using this test will give us a good point of view of process creation, execution and destruction performance.

Latency tests execution

The getppid, read, write, stat, fstat and open/close tests were conducted using the LMbench binary lat_syscall. The pagefault test was conducted using the LMbench binary lat_pagefault and the fork, exec and shell tests were conducted using the LMbench lat_proc binary.

lat_syscall has the following options

- -P : User can specify the degree of parallelism desired
- -W : User can specify the amount of time to wait before the tests begin. Allows the OS scheduler to complete other tasks before starting LMbench.
- -N : User can specify the number of times to repeat the test. This is almost always necessary as performing the test only once does not provide the user with an accurate estimate of the latency.
- file : User can specify a file to read, write, stat, fstat and open/close.

lat_pagefault has the following options

- -P : User can specify the degree of parallelism desired

- -W : User can specify the amount of time to wait before the tests begin. Allows the OS scheduler to complete other tasks before starting LMbench.
- -N : User can specify the number of times to repeat the test. This is almost always necessary as performing the test only once does not provide the user with an accurate estimate of the latency.
- file : User can specify a file of varying size to pagefault.

lat_proc has the following options

- -P : User can specify the degree of parallelism desired
- -W : User can specify the amount of time to wait before the tests begin. Allows the OS scheduler to complete other tasks before starting LMbench.
- -N : User can specify the number of times to repeat the test. This is almost always necessary as performing the test only once does not provide the user with an accurate estimate of the latency.

The option '-N 1001' was used across all the latency tests. '/data/app/lmbench/-foo' was an extra option used for lat_syscall while '/data/app/lmbench/foo_10MB' and 'foo_1MB' were extra options used for lat_pagefault.

With the aforementioned options the tests were automated using a shell script which ran each test ten times. While the tests were running, the phone screen was locked i.e turned off so we the overhead of keeping the display and sensors active was excluded. The script saved the results of each iteration and finally calculated the mean of each test which was then parsed and the passed on to the data visualizer.

Bandwidth tests

The bandwidth tests focus on memory access throughput i.e how much data can the OS move in a unit time. Memory performance is fundamental to the operation of the system [cite LMbench-usenix]. Using LMbench we focused on physical memory performance and inter process communication bandwidth using Unix pipes. The bandwidth tests covered by LMbench are as follows:

- **memory read:** This test measures how fast data is read into the processor. The processor computes the sum of an array of integer values by accessing every fourth word from the integer array in memory.

- **memory write:** This test measures how fast data is written into memory. The processors assigns an integer value to an array in memory by writing to every fourth word.
- **memory read and write:** This test combines the memory read and write operations. This is a commonly used pair of operations in a working system, therefore measuring the bandwidth of such an operation gives an overall estimate of the memory performance.
- **memory copy:** This test measures how fast data can be copied from a source to a destination in memory. It does a simple array copy such as destination[n]=source[n].
- **memory bzero:** This operation measures how fast the system can write zeros to a specified n bytes of memory. It's a basic memory operation that is used by applications such as dd.
- **memory bcopy:** This operation measures how fast the system can copy a specified n bytes of data from a source to destination in memory. This is an operation where certain architectures have optimized special instructions for bcopy.
- **memory pipe:** The memory pipe test involves measuring the performance of the Unix pipe between two processes. It moves 10MB of data as 64KB chunks through the Unix pipe.

Bandwidth test execution

The memory read, write, read and write, copy, bzero and bcopy tests were conducted using the LMbench binary bw_mem. The memory pipe test was executed using the LMbench binary bw_pipe.

bw_mem has the following options

- -P : User can specify the degree of parallelism desired
- -W : User can specify the amount of time in seconds to wait before the tests begin. Allows the OS scheduler to complete other tasks before starting LMbench.
- -N : User can specify the number of times to repeat the test. This is almost always necessary as performing the test only once does not provide the user with an accurate estimate of the bandwidth.
- size : User can specify the amount of memory the test program is to work on. It can be in kilobytes or megabytes.

bw_pipe has the following options

- -P : User can specify the degree of parallelism desired
- -W : User can specify the amount of time in seconds to wait before the tests begin. Allows the OS scheduler to complete other tasks before starting LMbench.
- -N : User can specify the number of times to repeat the test. This is almost always necessary as performing the test only once does not provide the user with an accurate estimate of the bandwidth.
- -m : User can specify the size of each message in bytes that are passed between the processes using the Unix pipe.
- -M : User can specify the total size of bytes that are passed between the processes using the Unix pipe.

The option '-N 1001' was used across all the bandwidth tests. For bw_mem tests, 10MB was the total amount of data that had to be worked on. For bw_pipe we chose the message size to be the standard 64KB and the total data to be transferred as 10MB.

With the aforementioned options the tests were automated using a shell script which ran each test ten times. While the tests were running, the phone screen was locked i.e turned off so the overhead of keeping the display and sensors active were excluded. The script saved the results of each iteration and finally calculated the mean of each test which was then parsed and passed along to the data visualizer.

7.6.2 Hackbench

Hackbench[6] is a widely used benchmarking or stress tester for the Linux kernel scheduler. It was originally written by Rusty Russell. Yanmin Zhang, Ingo Molnar and David Sommerseth were later contributors. Yanmin Zhang introduced the option of choosing between processes or threads at run time. Hackbench is a synthetic benchmark.

Scheduler test

The main task Hackbench accomplishes is creating a certain number of pairs of Schedulable threads or processes that communicate a specified number of messages between each other using either sockets or pipes. It then measures how long it takes for each Schedulable entity pair to send data between the sender and receiver.

We used Hackbench to compare the kernels scheduler performance of Honeydroid with Vanilla while using Unix sockets and pipes. We did not compare the performance between processes and threads.

Hackbench tests execution

Hackbench has a few options such as

- -p : Use the Unix pipe instead of the Unix socket(default option).
- -s : Specifies the payload size for each message transferred between sender/receiver.
- -l : Specifies the number of messages the sender sends to the receiver.
- -g : This option allows the user to input the number of groups of senders and receivers.
- -f : The user can tell the child processes/threads the number of file descriptors to use.
- -T : Use a POSIX thread instead of the default process.
- -P : The default option of using a process(fork()).

Hackbench was run 5 times with 360 tasks and communicated using pipes and sockets respectively. The mean value for the two scenarios was then calculated as the final result which was then parsed and sent to the data visualizer.

While using the tool we experienced that the options mentioned in the man page were not in sync with the code. So we had to fiddle around till we got the right options working. We also experienced an upper limit on the number of tasks that we could request Hackbench to create. The upper limit of tasks was 360, anything beyond that would cause Honeydroid to enter an unresponsive state which would then require a hard reboot to correct. We suspect this to be either a bug in L4Android/Fiasco.OC or a depletion of page table memory maintained by L4Android. The problem needs further investigation.

7.7 CPU Evaluation

The performance of the CPU was evaluated using two benchmarks. We used benchmarks that reported performance in terms of throughput (MIPs and MFLOPs) and response time. From [29] , we found that using MIPs and MFLOPs are highly dependent on the underlying architecture i.e the units are completely useless when comparing 32 bit processors with 64 bit processors or and ARM and Intel instruction set. They are useful in our case as we are using the same hardware but different software, therefore they are only indicative of any performance differences between vanilla and Honeydroid

and not absolute values. For evaluating the response time of the CPU we used a simple benchmark that measures how long the CPU takes to compute a specific number of digits in Pi. The two benchmark programs used to evaluate the CPU are 0xbench and RealPi. 0xbench is actually a benchmark suite i.e it comprises of many benchmark programs while RealPi is a stand-alone Java based app that has benchmarks in Java and C++. The benchmarks for CPU evaluation in 0xbench are kernel benchmarks while RealPi is a synthetic benchmark.

0xbench 0xbench[10] is a fully open source comprehensive benchmarking app for Android. It covers a gamut of benchmarking tests that can be used to measure the performance of an Android smartphone. It covers the following areas of benchmarking

- **Arithmetic:** Using Linpack and Scimark2.
- **Web:** Using Sunspider Javascript engine tester.
- **GPU:** 2d/3d performance using various animation programs.
- **Garbage Collection:** Using a Java program called GCBench.
- **System calls:** Using LibMicro and BYTE UnixBench.

In our CPU performance evaluation we use the Arithmetic benchmark. The Web and Garbage collection benchmarks are used in the Application evaluation. GPU performance evaluation was not included as the challenges involved in addressing bottlenecks in GPU virtualization are complex and mandate a study of its own. Finally, we used LMbench instead of what 0xbench provided due to its simplicity and adaptability.

7.7.1 LINPACK

LINPACK[8] is a fairly old software library used to compute linear algebra equations. It was originally coded in Fortran by Jack Dongarra, Jim Bunch, Cleve Moler and Gilbert Stewart who intended to use the program to measure the performance of Supercomputers in the 1970's and 1980's. It is still in use today to compare supercomputers floating point operation performance. The version of LINPACK provided by 0xBenchmark is a Java version. Therefore, using such a program we will be able to see how the CPU performs computationally when instructed from a userland Java program that is converted to Dalvik bytecode due to Android's VM architecture. It will also be indicative of the Just-In-Time compiler performance of Android and Honeydroid.

LINPACK test

The mathematical problem that LINPACK solves is a dense 500x500 matrix of linear equations of the type Ax=b. The matrix is a randomly generated one while the right hand side(b) is constructed such that the solution has all components equal to one. The Gaussian elimination with partial pivoting process is relied upon to find a solution to the linear equations [cite netlib linpackjava here]. This problem has $\frac{2}{3}n^3 + n^2$ floating point operations where a floating point operations is a floating point add or a floating point multiply with 64 bit operands.

LINPACK test execution

The LINPACK test was executed using the 0xbench app. The test was repeated 10 times. After each run, the memory and cache were cleared to prevent cached operations from possibly improving the performance. The results were collected, parsed and then passed on to the data visualizer.

7.7.2 Scimark2

Scimark2[9] in 0xbench is a composite Java benchmarking application that measures the performance of mathematical computations that are used in scientific and engineering problems. Scimark2 comprises of five computational kernels: Fast-Fourier Transform, Gauss-Seidel relaxation, Sparse matrix multiply, Monte Carlo integration and dense LU factorization.

Scimark2 tests

The computational kernels are meant to test the performance of the Floating-point Unit of the CPU by executing applications on Android's Dalvik VM. The problem sizes are small so as to isolate memory hierarchy effects and focus on the VM or CPU problems instead. The computational kernels are described below.

- **Fast Fourier Transform (FFT):** This kernel executes complex arithmetic, shuffling, non-constant memory references and trigonometric functions as a one-dimensional forward transform on four thousand complex numbers. Bit-reversal is done in the first part while the second part does the actual N.log(N) steps.
- **Jacobi Successive Over-relaxation (SOR):** This kernel makes use of basic "grid averaging" memory access patterns on a 100x100 grid that is common in

finite difference applications. Each A(i,j) is given an average weighting of its four nearest neighbors.

- **Monte Carlo:** This integration calculates the approximate value of Pi by performing the integral of the quarter of a circle $\sqrt{1-x^2}$ on [0,1]. Points are chosen at random with respect to the unit square and the ratio of that is computed within the circle. Random-number generators, synchronized function calls and function in-lining are made use of by this kernel.
- **Sparse matrix multiply:** This kernel uses a 1000x1000 sparse matrix with 5000 non-zeros. The sparse matrix has a sparsity structure stored in compressed-row format. Each row has approximately 5 non-zeros evenly spaced between the first column and diagonal.
- **Dense LU matrix factorization:** This kernel utilizes partial pivoting while exercising linear algebra and dense matrix operations. The kernel is the right-looking version of LU with rank-1 updates.

Scimark2 tests execution

The Scimark2 suite was executed 10 times using the 0xbench app. After each run, the memory on the phone was cleared to prevent a memory crunch from deteriorating performance or memory cache from improving performance. After 10 execution results the mean value for each test was calculated as the final result which was then parsed and then passed on to the data visualizer.

7.7.3 Real Pi

The RealPi[21] benchmark for Android by GeorgieLabs is an app that can be used to test the CPU(Integer and Floating-point Unit) and memory performance of Android smartphones. It implements two popular algorithms that are used to compute N-digits of Pi while it implements two other algorithms to calculate the Nth digit of Pi.

RealPi tests

The algorithms implemented are shown below

- AGM+FFT formula : This algorithm makes use of Arithmetic Geometric Means and Fast-Fourier Transforms implemented in C++ to calculate N-digits of Pi. The implementation is based on Takuay Ooura's pi_fftc6 program.

- Machin's formula : John Machin developed the Machin formula that can also be used to calculate N-digits of Pi. This algorithm has been implemented using Java and its BigDecimal class. It is expected to be slower than the previous AGM+FFT formula.
- Gourdon's formula : What's unique about this algorithm is that it can calculate the Nth digit of Pi without spending computing power on preceding digits to the Nth position. This algorithm is better suited for testing the CPU than memory performance. The shortcoming of this program is it works only for $N > 50$. It is based on Xavier Gourdon's pidec program written in C++.
- Bellard's formula : This algorithm is based on Fabrice Bellards formula, it is similar to the previous algorithm but is used when $N <= 50$.

RealPi tests execution

Our evaluation using RealPi included the use of the AGM+FFT formula and Machin's formula. The reason we chose these two is because, the AGM+FFT is implemented in C++ while Machin's formula has been implemented in Java. This difference can be significant with respect to performance. The AGM+FFT was used to compute one million digits of Pi while Machin's formula was used to compute only five thousand digits of Pi. We repeated the test 10 times, clearing the memory after each iteration to prevent cached memory from affecting the performance. The mean value from 10 iterations was calculated to be the final result. The results were then parsed and passed to the data visualizer.

7.8 Application Evaluation

In this section we cover the different benchmarks that were used to evaluate the performance at the application level. The two main benchmark programs used here are Dalvik VM garbage collection and Sunspider. The garbage collection benchmark is a synthetic benchmark while Sunspider has toy benchmarks and synthetic benchmarks within it. The focus of these benchmarks is on the response time and not on the throughput.

7.8.1 Dalvik VM garbage collection

The original benchmark was written by John Ellis and Pete Kovac of Post Communications. Hans Boehm from Silicon Graphics modified it for Java. The benchmark was designed to model the properties of allocation requests that are significant to garbage

collection techniques [25]. The benchmark provides an overall performance number as well as a detailed estimate of the garbage collectors performance with varying object lifetimes.

Garbage collector tests

The garbage collector performance is vital to memory management thereby affecting the performance of the operating system. The benchmark stores two data structures throughout the lifetime of the test, this is to mimic a real applications which retains some live in-memory data. One of the data structures is a tree with many pointers and the other is a large array containing double precision floating point numbers. Both data structures are of comparable sizes. The benchmark then measures the time taken to allocate and collect balanced binary trees of different sizes trying to allocate the same amount of memory in each iteration. The time taken to allocate and collect balanced binary trees of different sizes are used as the performance results. The final time reported by this benchmark also includes times where the benchmark got scheduled or interrupted by a timer. Therefore, the state of the system affects the performance results which means that the results are not uniform.

Garbage collector tests execution

The garbage collector tests were executed 10 times using the 0xbench app. After each run, the memory on the phone was cleared to prevent a possibly memory crunch from altering the performance of the phone. After 10 executions, the mean value for each test was calculated as the result. The results were then parsed and sent to the data visualizer.

7.8.2 Sunspider

The version of Sunspider[15] bundled in 0xbench is 0.9.1. This is a version behind the latest which is 1.0. It's a benchmarking tool that can measure the performance of Javascript within a browser or the stand-alone Javascript engine.

Sunspider tests

The Sunspider benchmark focuses on testing Javascript and not the DOM or other browser APIs. Sunspider was designed to mimic real world scenarios and cover the different areas the Javascript language is used for. Sunspider covers the following cases

- **3d:** These test cases comprise of performing 3d operations on 3d objects such as cube rotation, 3d morphing and 3d ray tracing.
- **Memory access:** These test cases bundle operations such as creating a bottom-up binary tree, Fannkuch operation[11], nbody-access[12], n-sieve[13] benchmark which computes the number of prime numbers from 2 to m.
- **Bit operations:** This set consists of tests that perform bit level operations such as AND, OR, ADD and SHIFT to compute prime numbers or specified operations.
- **Control-flow:** This test case measure the time taken to perform recursive computations like computing the Fibonacci series.
- **Cryptographic protocols:** This suite involves using cryptographic ciphers such as AES to encrypt, decrypt plain-text messages using the Javascript DOM engine. It also measures the performance of the hashing algorithm md5 and sha1.
- **Date:** The date tests perform various date, time and calendar computations using the system clock. Such cases are used by browsers to display calendars or data and times for different time-zones.
- **Mathematical computations:** The mathematical tests implement algorithms such as the CORDIC Algorithm which is a calculator using simple bit operands such as add, multiply and shift, Partial sum algorithm and spectral normalization. These three algorithms test the computational speed of the Javascript engine.
- **Regular-expression matching:** This test tests how fast the Javascript engine can match regular expressions using its decision trees which will be testing the memory access of the underlying system.
- **String manipulation:** In this suite, Base64 scheme is tested i.e binary-to-text encoding/decoding. The FASTA scheme which is text format used to store DNA sequences and its quality score is benchmarked. Other tests include tag clouds, Javascript code unpacking and string validation.

Sunspider tests execution

Executing the Sunspider test from 0xbench runs each Javascript test 5 times and finally reports the mean value of each test result along with a tolerance. The results from 0xbench were directly used Sunspider did all the calculations for us. The results were parsed and then sent over to the data visualizer.

7.9 Power Evaluation

Power evaluation is a very complex task. Based on the load of the system the OS can be in a number of states. This makes power measurement results non-uniform. To account for the aforementioned reason we restricted our power evaluation to only two scenarios:Idle state and Voice Call. We regard these two scenarios to be practical, identical to real life situations and indicative of any overheads such as virtualization.

We measure the current drawn by the phone(without the battery inserted in the phone) by connecting an Agilent U1253B Multimeter in series with the phone and Agilent DC Power Supply E3645A on the positive line. The multimeter logs the current being drawn by the phone every second. The DC power supply maintain a 4.3V potential difference while supplying a varying current up to 2.2A. A Deutsche Telekom SIM card was used to connect to the GSM and GPRS cellular network.

7.9.1 Idle Test and execution

In this test the phone is left in an idle state which is defined as "the state in which the phone is connected to a cellular network base station with a strong signal strength(between -79dB and -50dB), the touch screen is switched off, the phone is silent and only necessary and/or default processes are running in the background". The measurements were made for 30 minutes and repeated 5 times during the course of a day. Since the signal strength of the base station tower changes with traffic intensity the measurements were made across business hours which are practical in all business purposes. The mean value of all repetitions was then considered to be the idle current drawn by the phone.

7.9.2 Voice Call Test and execution

A voice call test is defined as "Sixty second of idle time, after which the phone rings in the silent mode, only the touch screen lights up for five seconds, after which a voice conversation(a Steve Jobs speech at Stanford plays through earphones kept next to the phone microphone) occurs for ten minutes, followed by sixty to eighty seconds of idle time all while the phone has a strong signal strength(between -79dB and -50dB)". We found it hard to maintain an absolute test definition during test repetitions due to the phone responsiveness and general multitasking issues. We were able to fall within a ten second range of the definition. The measurements were repeated 5 times during the course of a business working day to reflect signal strength changes of base stations. The mean value of all repetitions was then considered to be the current drawn by the phone during a voice call.

8 Evaluation Results

8.1 Performance Results and Analyses

The results and analyses of the performance evaluation of Honeydroid and Vanilla are described in the following sections. First, the results of the performance evaluation are reported after which the results are analyzed carefully. The results have been tabulated in this section for quick reference. They have also been visually represented as either histograms, or line curves as deemed appropriate for effective visual understanding.

8.2 Networking

The results obtained are shown in 8.1, 8.2, 8.3 and 8.4. The TCP and UDP throughput are reported in Kilo bits per second(Kbps) and the data transfer sizes are reported in Kilo Bytes(KB). The results have been tabulated according to the Transport Protocol and MTU size. Each table enumerates the total data transferred and the direction of the transfer, along with the throughput of Vanilla and Honeydroid.

8.2.1 Results

From Table 8.1 it is evident that the down-link performance of Honeydroid is comparable to that of Vanilla. Honeydroid has the highest down-link throughput of 48.4Kbps for the 1024KB data transfer and the least down-link throughput of 46.5Kbps for the 128KB data transfer. Honeydroid has the highest up-link throughput of 32.6Kbps for the 128KB data transfer and the least up-link throughput of 30.6Kbps for the 2176KB data transfer. Vanilla has the highest down-link throughput of 48.3Kbps for the 1024KB data transfer and the least down-link throughput of 46.9Kbps for the 512KB data transfer. Vanilla has the highest up-link throughput of 31.3Kbps for the 256KB data transfer and the least up-link throughput of 30.7Kbps for the 2176KB data transfer.

From Table 8.2 we can see that the down-link performance of Honeydroid is comparable to Vanilla. Honeydroid has the highest down-link performance of 52.1Kbps for the 1024KB data transfer and the least down-link throughput of 51.4Kbps for the 2176KB and 512Kb data transfer. Honeydroid has the highest up-link throughput for 35.4Kbps

TCP & MTU=604 bytes		
KB transferred and Direction	Vanilla(Kbps)	Honeydroid(Kbps)
128KB down-link	47.5	46.5
128KB up-link	32.0	32.6
256KB down-link	47.9	47.5
256KB up-link	31.3	31.5
512KB down-link	46.9	47.5
512KB up-link	31.1	31.5
1024KB down-link	48.3	48.4
1024KB up-link	30.8	30.8
2176KB down-link	47.9	47.7
2176KB up-link	30.7	30.6

Table 8.1: TCP Throughput of Vanilla and Honeydroid with MTU=604 bytes

for the 128KB data transfer and the least up-link throughput of 32.0Kbps for the 2176KB data transfer. Vanilla has the highest down-link throughput of 52.1Kbps for the 256KB data transfer and the least down-link throughput of 50.8Kbps for the 512KB data transfer. Vanilla has the highest up-link throughput of 34.6Kbps for the 128KB data transfer and the least up-link throughput of 32.7Kbps for the 2176KB data transfer.

Table 8.3 the total data transferred was 1025KB. Honeydroid has a down-link throughput of 52.7Kbps and up-link throughput of 32.1Kbps. Vanilla has a down-link throughput of 52.0Kbps and up-link throughput of 34.2Kbps.

Table 8.4 the total data transferred was 1025Kb. Honeydroid has a down-link throughput of 53.6Kbps and up-link throughput of 35.6Kbps. Vanilla has a down-link throughput of 54.7Kbps and up-link throughput of 35.8Kbps.

8.2.2 Analysis

Tables 8.1 and 8.2 depict the TCP throughput for the various MSS and data transfer sizes. It is clear that the down-link capacity of the GPRS connection is much better with a smaller MSS[48], this is a known fact about GPRS. The up-link capacity is not as good with a lower MSS unlike the down-link throughput. This can be attributed to the MSS and the fact that when checking up-link capacity, Honeydroid does not need to buffer as much data as it has to while acting as the server, which involves receiving and acknowledging data received. Although the GPRS tests do not highlight anything new, the framework developed can be applied to new versions of Honeydroid that have

TCP & MTU=1500 bytes		
KB transferred and Direction	Vanilla(Kbps)	Honeydroid(Kbps)
128KB down-link	51.1	51.9
128KB up-link	34.6	35.4
256KB down-link	52.1	51.4
256KB up-link	33.7	33.3
512KB down-link	50.8	51.4
512KB up-link	33.3	33.3
1024KB down-link	51.5	52.1
1024KB up-link	32.8	32.8
2176KB down-link	51.5	51.4
2176KB up-link	32.7	32.0

Table 8.2: TCP Throughput of Vanilla and Honeydroid with MTU=1500 bytes

UDP & MTU=604 bytes		
KB transferred and Direction	Vanilla(Kbps)	Honeydroid(Kbps)
1025KB UDP down-link	52.0	52.7
1025KB UDP up-link	34.2	32.1

Table 8.3: UDP Throughput of Vanilla and Honeydroid with MTU=1500 bytes

UDP & MTU=1500 bytes		
KB transferred and Direction	Vanilla(Kbps)	Honeydroid(Kbps)
1025KB UDP down-link	54.7	53.6
1025KB UDP up-link	35.8	35.6

Table 8.4: UDP Throughput of Vanilla and Honeydroid with MTU=1500 bytes

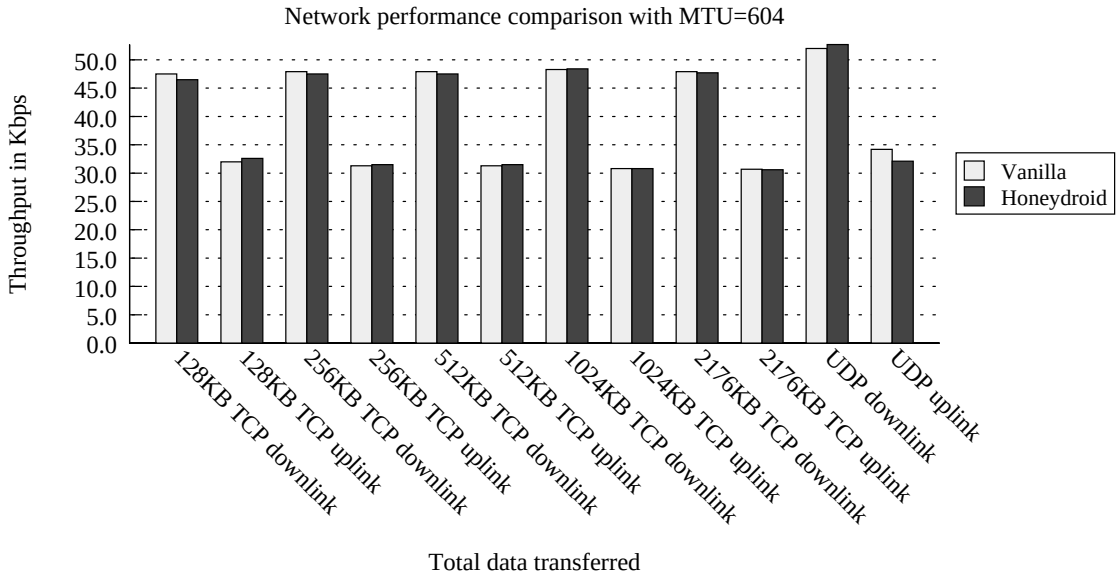

Figure 8.1: Vanilla and Honeydroid comparison of Network performance with MTU=604B

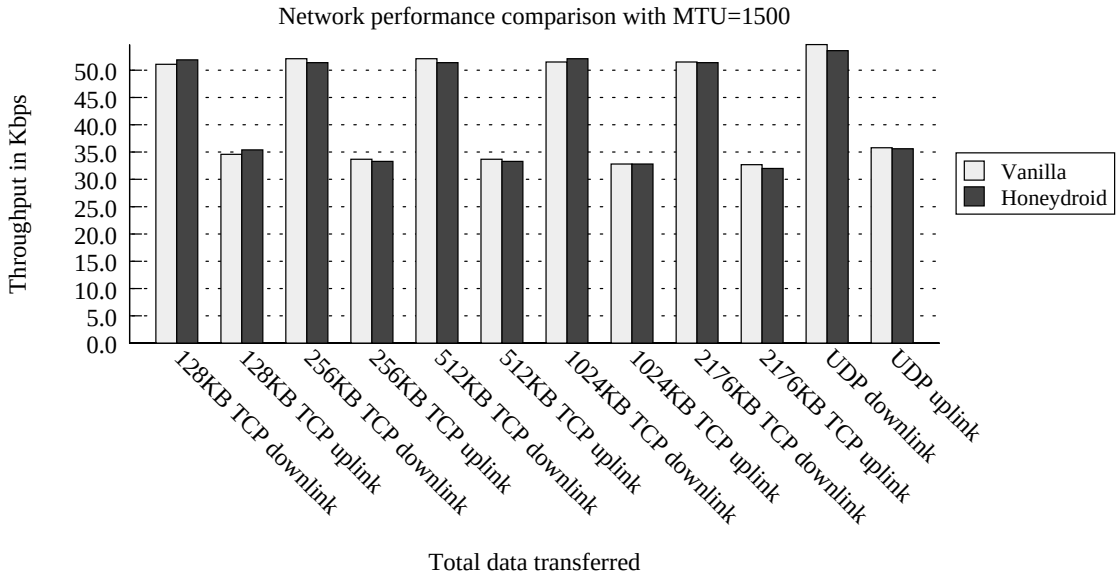

Figure 8.2: Vanilla and Honeydroid comparison with MTU=1500B

System call latencies		
System call	Vanilla(μs)	Honeydroid(μs)
getppid()	0.20243	6.00573
read()	0.48684	6.74444
write()	0.4037	6.5369
stat()	4.3827	10.5638
fstat()	1.1488	6.9204
open()/close()	6.0714	20.4772
Pagefault on 10MB	0.27484	6.07582
Pagefault on 1MB	2.4119	60.5706
Process fork()+exit()	384.6667	14134.4538
Process fork()+execv()	384.6667	14134.4538
Process fork()+/bin/sh	400.8462	14199.9917

Table 8.5: System call latency of Vanilla and Honeydroid

Wifi or 4G/LTE.

8.3 Operating System Results

The results and analyses for the different OS tests are documented in the following subsections.

8.3.1 LMbench

After the shell script executed the test cases, a final results file was created for Vanilla and Honeydroid. The results obtained from that file are reported in the following subsection.

Results

The results obtained from running LMbench are shown in Tables 8.5 and 8.6. Table 8.5 shows the results from the latency tests. The latency of the system calls were measured and reported in microseconds. Table 8.6 has the results from the bandwidth tests. The bandwidth was measured in Megabytes per second it took the system to operate on 10.49 Megabytes of data in physical memory.

From Table 8.5 it is evident that the system call latencies of Honeydroid are magnitudes higher. A simple getppid system call takes only .20243 μs in Vanilla but takes

System call bandwidths		
System call	Vanilla(MB/s)	Honeydroid(MB/s)
memory read	1020.974	932.779
memory write	686.054	649.854
memory read&write	606.947	576.851
memory copy	380.04	358.454
memory bzero	1598.257	1585.366
memory bcopy	647.226	608.673
Unix pipe	490.627	477.14

Table 8.6: Memory bandwidth of Vanilla and Honeydroid in moving 10.49MB

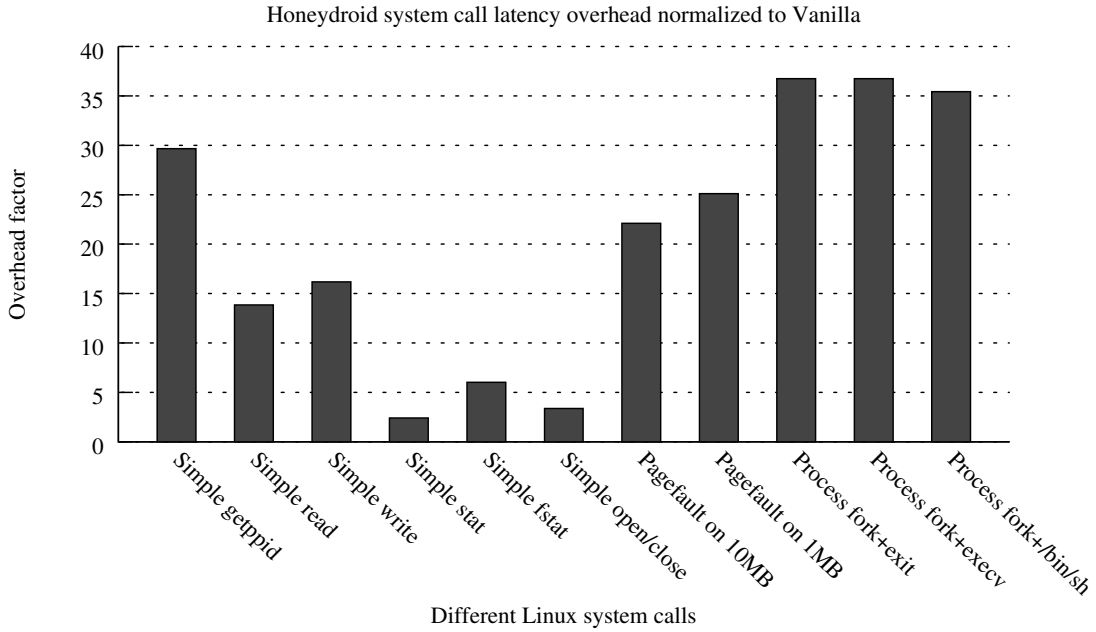

Figure 8.3: Vanilla and Honeydroid comparison of system calls latency

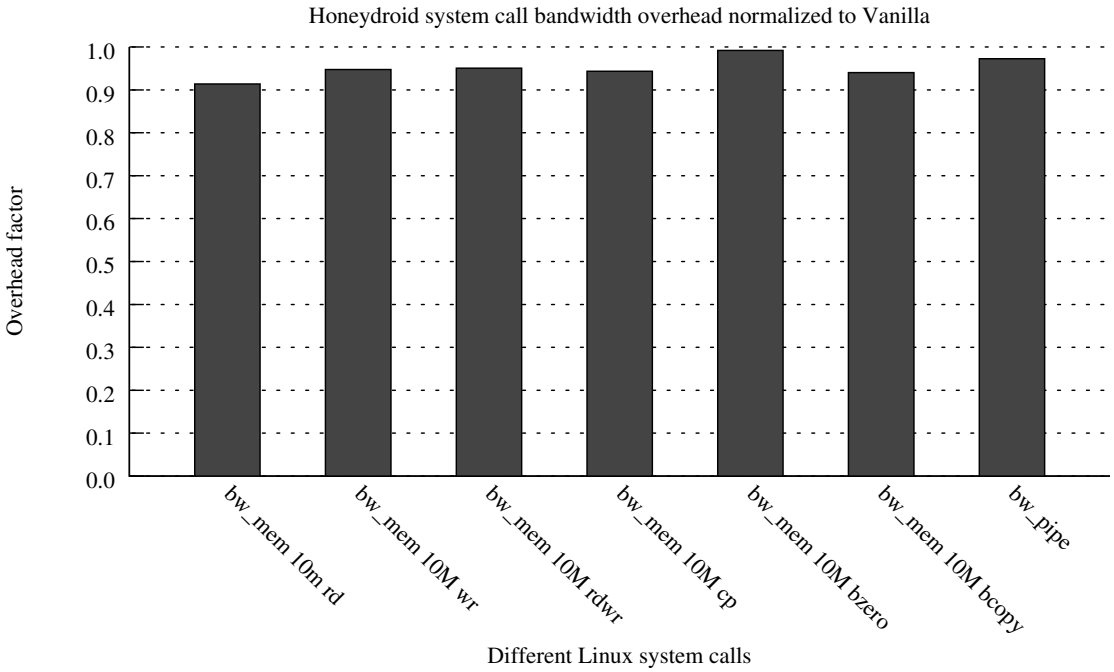

Figure 8.4: Vanilla and Honeydroid comparison of system calls bandwidth

6.00573 μs in Honeydroid. The getppid is the system call with least latency in both vanilla and Honeydroid. The most expensive system call is the process fork()+/bin/sh system call. It took 400.8462 μs on Vanilla and 14199.9917 μs on Honeydroid.

From Table 8.6 we can see that the slowest operation is the memory copy which is 380.04 MB/s on Vanilla while it is 385.454 MB/s on Honeydroid. The fastest memory operation is the bzero operation which is 1598.257 MB/s on Vanilla and 1585.366 MB/s.

Analysis

To help us analyze the data in a more visual form we plotted the system call latency of Honeydroid normalized to Vanilla in Figure 8.3. In the graph we can see that the stat and open/close system calls are the least affected by para-virtualization. The system calls that are hurt the most are process forks, exits and execvs.

The file system latencies do not incur as much overhead as the other system calls do due to the fact that accessing memory within ones Task(Protection Domain) is not expensive versus accessing memory across Tasks in L4Android/Fiasco.OC.

The process fork tests are very expensive operations in Honeydroid. This can be attributed to the following reasons: Kernel entry and exit involves not only CPU privilege switch but also exception handling, page table flushes, address space switches and IPC messaging which are all time consuming operations. In the test that measures the time to fork and exit, Honeydroid performs badly. This can be explained by the memory virtualization mechanism of L4Android[36]. Whereas in Vanilla, these complexities are not present therefore incurring a lower latency than Honeydroid.

The page faults are expensive operations in both Honeydroid and Vanilla but the overhead incurred by Honeydroid can be explained by the fact that page tables for Honeydroid are maintained as shadow page tables in L4Android. The reason for using a shadow page table which is known to be expensive[24] is because the ARM Cortex-A9 architecture does not support hardware virtualization. Therefore, page table management becomes a bottleneck for page faults in Honeydroid.

The results from the memory bandwidth tests visualized in Figure 8.4 show that Honeydroid has a maximum of 10% overhead. The graph conveys a good point about micro-kernel based systems which is, memory operations incur a very small overhead. The reason this observation was possible was due to the test LMbench performs, it simply unrolls a loop that sums up integer values. Such an operation hardly needs the kernel services and branch prediction, thereby testing the performance of memory operations. Therefore, it is safe to say that operations that do not require kernel services have a small performance impact due to para-virtualization.

From the results of LMbench we can conclude that the response time performance of

Hackbench results		
Hackbench communication	Vanilla(s)	Honeydroid(s)
Unix Socket	2.4444	5.4505
Unix Pipe	2.3206	6.149

Table 8.7: Hackbench performance of Vanilla and Honeydroid

system call operations on Honeydroid is orders of magnitude higher than native Android. This is due to the virtualization overhead caused by the kernel entry/exits, address space switches and shadow page table management during system call processing. Memory accesses on Honeydroid performs almost as efficiently as it does on native Android. This is because no kernel services are needed during execution. This also implies that by avoiding kernel services the virtualization overhead can be eliminated.

Even though the system call performance of Honeydroid is an order of magnitude higher than Vanilla, the overall system performance may be different. And this is what the thesis investigates in further sections.

8.3.2 Hackbench

After Hackbench was executed 5 times, the mean value was calculated. The results for Vanilla and Honeydroid are reported in the following subsection.

Results

The results obtained from running Hackbench are shown in Table 8.7. The time taken for Vanilla using pipes was 2.3206s and for Honeydroid it was 6.149s. Vanilla took 2.4444s and Honeydroid took 5.4505s using sockets instead of pipes.

Analysis

From Figure 8.5 it is clear that Honeydroid takes more than double the time of Vanilla. This virtualization overhead is due to the costly affair of task destruction in Fiasco.OC. In Table 8.5 we saw that process fork()+exit() for Honeydroid was a little more than 35 times slower. Since Hackbench creates and destroys 360 processes, this causes page faults and page table invalidations during task destruction that results in an overhead. We did not investigate why and how the overhead factor was not in-line with what was observed in Table 8.3 but we suspect the load on the scheduler for the Hackbench tests

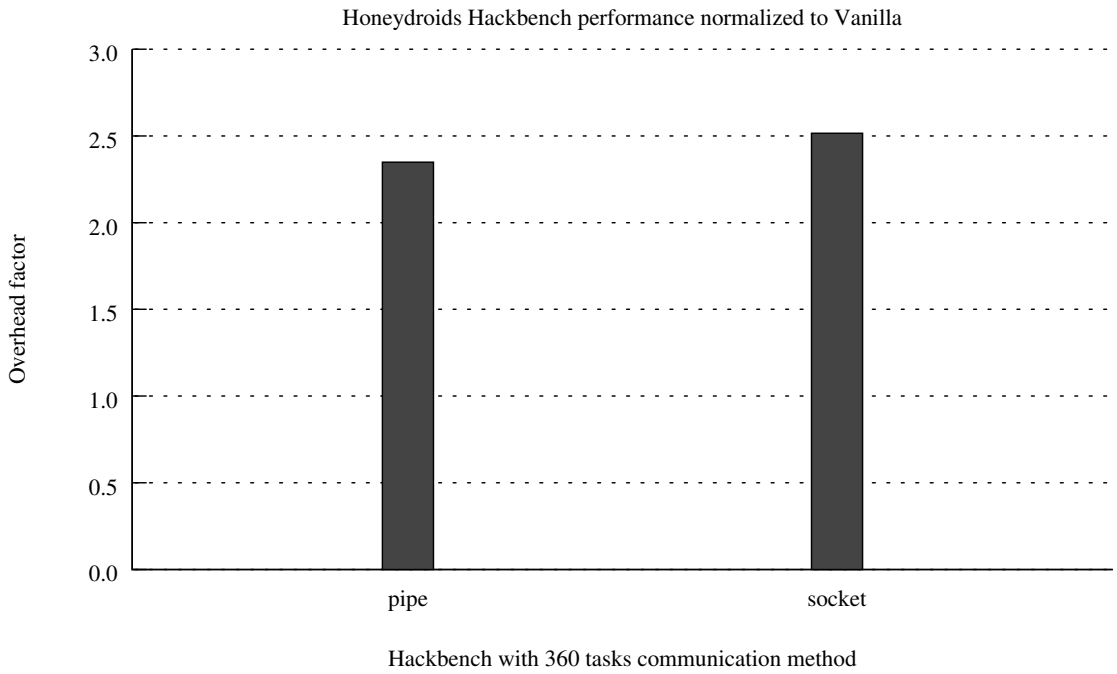

Figure 8.5: Vanilla and Honeydroid comparison of Hackbench Sockets and Pipes

LINPACK results	
Vanilla(MFLOPs/s)	Honeydroid(MFLOPs/s)
43.0212685091812	41.7586996895219

Table 8.8: LINPACK performance of Vanilla and Honeydroid

introduced an anomalous behavior in Honeydroid which resulted in an overhead less than the process fork()+exit() overhead which is the worst case path.

8.4 CPU Results

The results and analyses for the different CPU tests are documented in the following subsections.

8.4.1 LINPACK

After LINPACK was executed 10 times, the mean value was calculated. The results for Vanilla and Honeydroid are reported in the next subsection.

Results

The results obtained from running LINPACK are shown in Table 8.8.

Analysis

From Figure 8.6 we can see that Honeydroid is very close to the performance of Vanilla for the LINPACK test. Initially we thought it was because Fiasco.OC schedules floating-point instructions lazily but that was overturned by a higher possibility of incomplete utilization of the Floating-point Unit(FPU) on the hardware. It could also be due to cache and Translation Lookaside Buffer(TLB) misses. Without further investigation we cannot be sure of what caused the overhead.

8.4.2 Scimark2

After Scimark2 was executed 10 times, the mean value was calculated. The results for Vanilla and Honeydroid are reported in the following subsection.

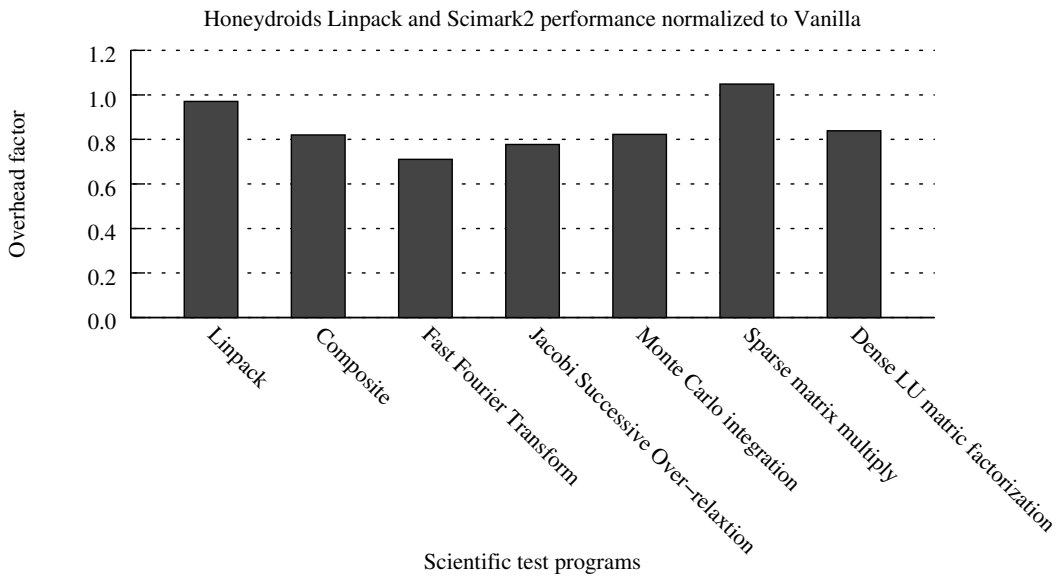

Figure 8.6: Vanilla and Honeydroid comparison for LINPACK

Results

The results obtained from running Scimark2 using 0xbench are shown in Table 8.9.

Analysis

From Figure 8.6 we can see that Honeydroid takes an approximately 20% performance hit when compared with Vanilla. The 20% overhead can be ascribed to L4Android/Fiasco.OC not making full use of the FPU. Another possible explanation for the reduced performance in Honeydroid is due to cache and TLB misses, the two are hard to separate and we suspect that the misses are a likely reason for negative performance impact seen in Honeydroid. The 30% drop in the Fast-Fourier Transform test is also most likely due to the underutilization of the FPU as we expect the performance of computational tests to be almost the same if not slightly less like that of the LINPACK result. What was most fascinating about the Scimark2 results was possibly the surfacing of a phenomenon known as Bldy's anomaly [23] in the Sparse matrix multiply test for Honeydroid wherein, reducing the number of page frames results in a decrease in the number of page faults thereby improving performance.

Scimark2 results		
Kernel	Vanilla(MFLOPs/s)	Honeydroid(MFLOPs/s)
Composite	57.9005765114589	47.4804130266808
Fast-Fourier Transform	40.7165563055124	28.9254027166886
Jacobi SOR	132.095854462908	102.664294084119
Monte Carlo	10.7026968514318	8.80610360078175
Sparse matrix multiply	38.5942264426247	40.4635720336805
Dense matrix LU factorization	67.3935484948160	56.5426926981346

Table 8.9: Scimark2 performance of Vanilla and Honeydroid

RealPi results		
Pi Algorithm and No. of Pi digits	Vanilla(s)	Honeydroid(s)
AGM+FFT 1 million digits	24.08	308.17
Machin's formula 5000 digits	14.546	18.66

Table 8.10: RealPi Bench performance of Vanilla and Honeydroid

8.4.3 RealPi Bench

After RealPi Bench was executed 10 times, the mean value was calculated. The results for Vanilla and Honeydroid are reported in the following subsection.

Results

The results obtained from running RealPi Bench are shown in Table 8.10. The time taken for Vanilla to compute one million digits of Pi was 24.08s while Honeydroid took 308.17s while the time to compute five thousand digits of Pi using Machin's formula took 14.546s for Vanilla and 18.66s for Honeydroid.

Analysis

From Figure 8.7 it is evident that Honeydroid takes close to 14 times that of Vanilla when using the C++ based AGM+FFT benchmark. This result is very peculiar as in the Scimark2 test result from Table 8.9, the Monte Carlo kernel also calculates digits in Pi and the overhead incurred there was only 19%. We suspected the large overhead here to be similar to what was mentioned in 8.4.2 where the FPU was not used. In order to demonstrate the validity of our suspicion we ran the RealPi bench using Machin's

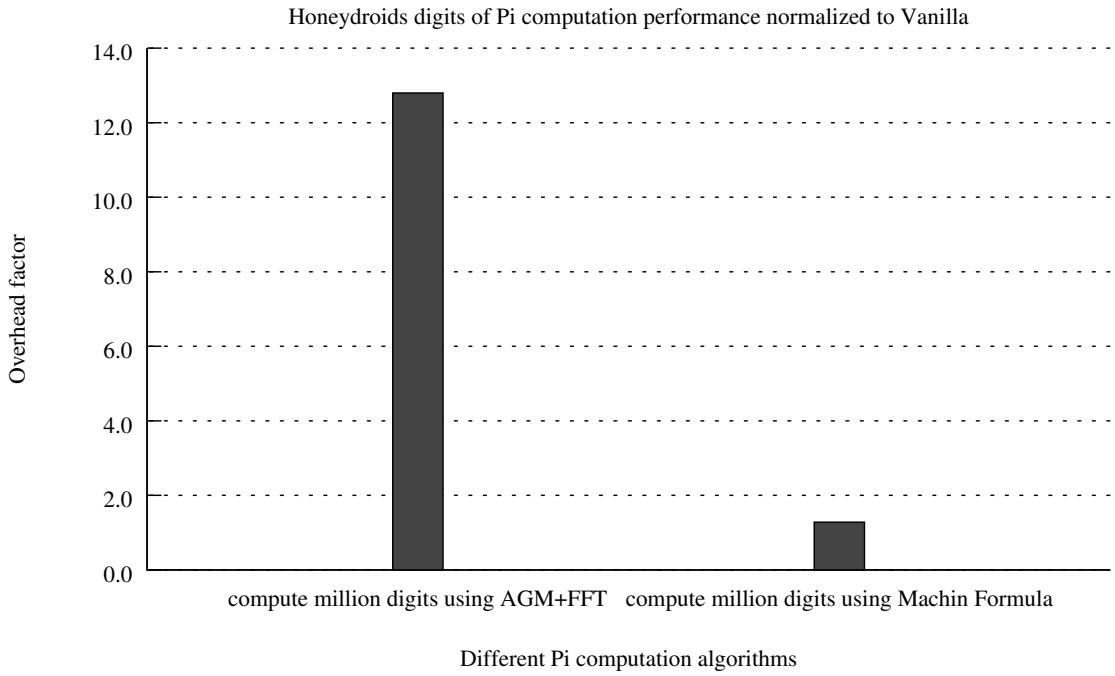

Figure 8.7: Vanilla and Honeydroid comparison

Dalvik VM Garbage Collector results		
No. of trees and depth	Vanilla(ms)	Honeydroid(ms)
37448 top-down trees of depth 2	388.1	376.1
37448 bottom-up trees of depth 2	440.6	386.8
8456 top-down trees of depth 4	473.5	386.1
8456 bottom-up trees of depth 4	452.6	336.0
2064 top-down trees of depth 6	444.9	366.9
2064 bottom-up trees of depth 6	437.9	364.6
512 top-down trees of depth 8	472.1	366.7
512 bottom-up trees of depth 8	464.3	366.4
Total time	4123.7	4016.2

Table 8.11: Dalvik VM Garbage collection performance of Vanilla and Honeydroid

formula which is programmed in Java. From Figure 8.7 we can see that the performance of Honeydroid improved significantly when compared with the AGM+FFT implementation which required certain floating-point operations that was used by the Java based Machin's formula program. Therefore, if Honeydroid is presented with the full use of the FPU then floating-point computations and memory operations will experience a very small performance overhead.

8.5 Application Results

The results and analyses for the different application tests are documented in the following subsections.

8.5.1 Dalvik VM Garbage Collector

After the garbage collector benchmark was executed 10 times, the mean value was calculated. The results for Vanilla and Honeydroid are reported in the following subsection.

Results

The results obtained from running the garbage collector benchmark using 0xbench are shown in Table 8.11. The total time taken for Vanilla is 4123.7 ms while Honeydroid took only 4016.2 ms.

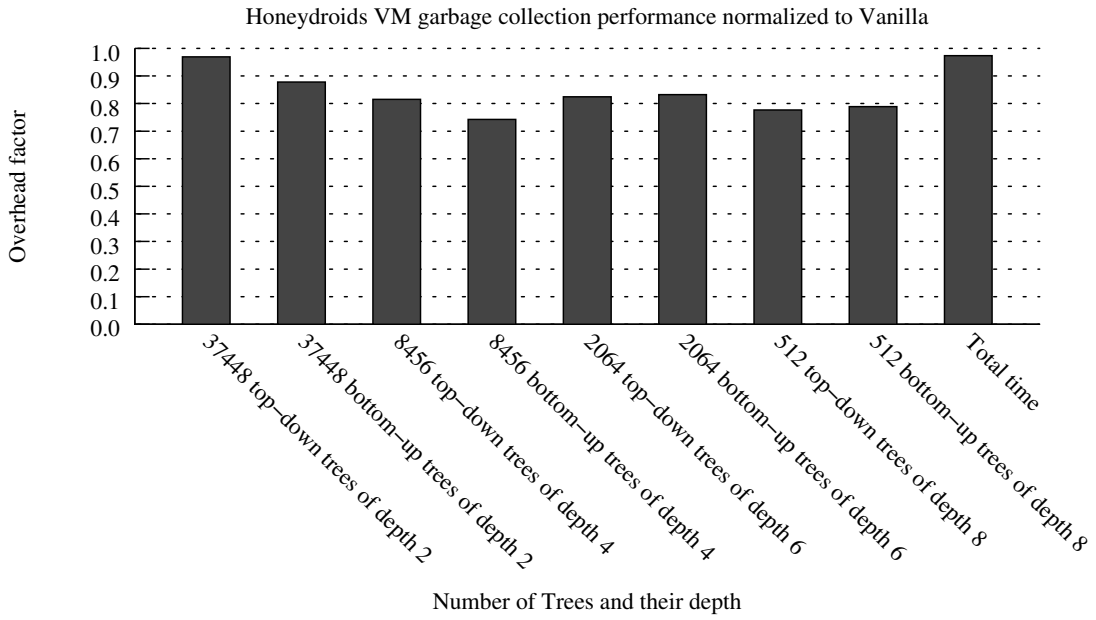

Figure 8.8: Vanilla and Honeydroid comparison of Dalviks VM garbage collector

Analysis

From Figure 8.8 it is evidently non-intuitive that Honeydroid performs better than Vanilla in most tests while the total time taken by Honeydroid and Vanilla are almost the same. Honeydroid appears to be faster than Vanilla by approximately 20% in all cases except for the 37448 top-down trees of depth 2 test. This improved performance seems rather peculiar when compared to the system call bandwidth results in Table 8.6. We suspect that the garbage collector test introduces some scheduling anomaly in L4Android, so that the garbage collector is serviced faster than in Vanilla. The matter requires further investigation as it brings up an important and previously unknown behavior to the operating systems community. Using hardware performance counters and branch prediction, investigators can gain some insight into this unusual behavior. But considering the overall performance of the garbage collector we can see it correlates with the bandwidth test results from Table 8.6 and Figure 8.4.

8.5.2 Sunspider

Sunspider through 0xbench runs each test 5 times and provides results with a 95% confidence level. This is a very useful aspect of Sunspider which is not provided by most benchmarking tools. The results are shown in the following subsection.

Results

The results obtained from running the Sunspider benchmark using 0xbench are shown in Tables 8.12 and 8.13. The total time taken for Vanilla is 3019.3ms while Honeydroid took only 3669.3ms with a tolerance of approximately 1.2% for Vanilla and 1% for Honeydroid.

Analysis

In Figure 8.9, we can see that the total overhead Honeydroid displays is just a little more than 20%. Considering the overall performance of the 3d, access, bitops, control-flow, security and date tests we see that they fall under 20% overhead. This can be considered to be the virtualization overhead. This benchmark was run for only the Javascript engine, we could expect the performance drop to be higher if the Javascript engine was tested from within the web browser. The mathematical tests experienced an approximate 30% overhead. This is most likely due to reasons explained in 8.4.2 but we are not 100% confident of that. Cache pressure could be another culprit for the performance overhead observed here. The performance overhead indicated by the

Sunspider results		
Sunspider test	Time (ms)	Tolerance
3d:	473.3	+/- 2.7%
cube:	199.4	+/- 8.0%
morph:	135.2	+/- 1.4%
raytrace:	138.7	+/- 4.7%
access:	314.9	+/- 6.7%
binary-trees:	14.8	+/- 11.8%
fannkuch:	133.0	+/- 0.6%
nbody:	132.9	+/- 15.6%
nsieve:	34.2	+/- 5.1%
bitops:	220.7	+/- 7.9%
3bit-bits-in-byte:	21.8	+/- 1.4%
bits-in-byte:	33.6	+/- 1.1%
bitwise-and:	73.8	+/- 4.7%
nsieve-bits:	91.5	+/- 19.2%
controlflow:	13.1	+/- 1.7%
recursive:	13.1	+/- 1.7%
crypto:	192.5	+/- 2.9%
aes:	90.1	+/- 4.6%
md5:	54.7	+/- 5.0%
sha1:	47.7	+/- 5.3%
date:	400.2	+/- 7.5%
format-tofte:	162.5	+/- 17.4%
format-xparb:	237.7	+/- 9.9%
math:	264.3	+/- 12.9%
cordic:	129.5	+/- 25.8%
partial-sums:	100.1	+/- 3.2%
spectral-norm:	34.7	+/- 4.2%
regexp:	96.9	+/- 1.1%
dna:	96.9	+/- 1.1%
string:	1043.4	+/- 1.4%
base64:	105.0	+/- 14.7%
fasta:	174.8	+/- 1.6%
tagcloud:	205.8	+/- 2.7%
unpack-code:	436.8	+/- 1.8%
validate-input:	121.0	+/- 2.2%
Total:	3019.3	+/- 1.2%

Table 8.12: Sunspider results for Vanilla

Sunspider results		
Sunspider test	Time (ms)	Tolerance
3d:	573.3	+/- 3.1%
cube:	244.8	+/- 7.9%
morph:	164.7	+/- 3.1%
raytrace:	163.8	+/- 3.4%
access:	370.0	+/- 3.8%
binary-trees:	17.0	+/- 10.9%
fannkuch:	160.9	+/- 1.1%
nbody:	148.6	+/- 9.7%
nsieve:	43.5	+/- 8.6%
bitops:	258.0	+/- 10.0%
3bit-bits-in-byte:	25.2	+/- 5.1%
bits-in-byte:	39.7	+/- 3.8%
bitwise-and:	84.2	+/- 4.9%
nsieve-bits:	108.9	+/- 20.5%
controlflow:	15.4	+/- 2.4%
recursive:	15.4	+/- 2.4%
crypto:	220.9	+/- 2.3%
aes:	101.4	+/- 3.7%
md5:	63.6	+/- 6.2%
sha1:	55.9	+/- 5.7%
date:	477.6	+/- 7.7%
format-tofte:	173.0	+/- 15.1%
format-xparb:	304.6	+/- 11.7%
math:	340.1	+/- 13.9%
cordic:	166.1	+/- 26.0%
partial-sums:	129.6	+/- 4.7%
spectral-norm:	44.4	+/- 4.6%
regexp:	146.4	+/- 1.0%
dna:	146.4	+/- 1.0%
string:	1267.6	+/- 1.5%
base64:	140.4	+/- 2.3%
fasta:	205.2	+/- 3.3%
tagcloud:	249.1	+/- 2.9%
unpack-code:	519.0	+/- 2.4%
validate-input:	153.9	+/- 4.5%
Total:	3669.3	+/- 1.0%

Table 8.13: Sunspider results for Honeydroid

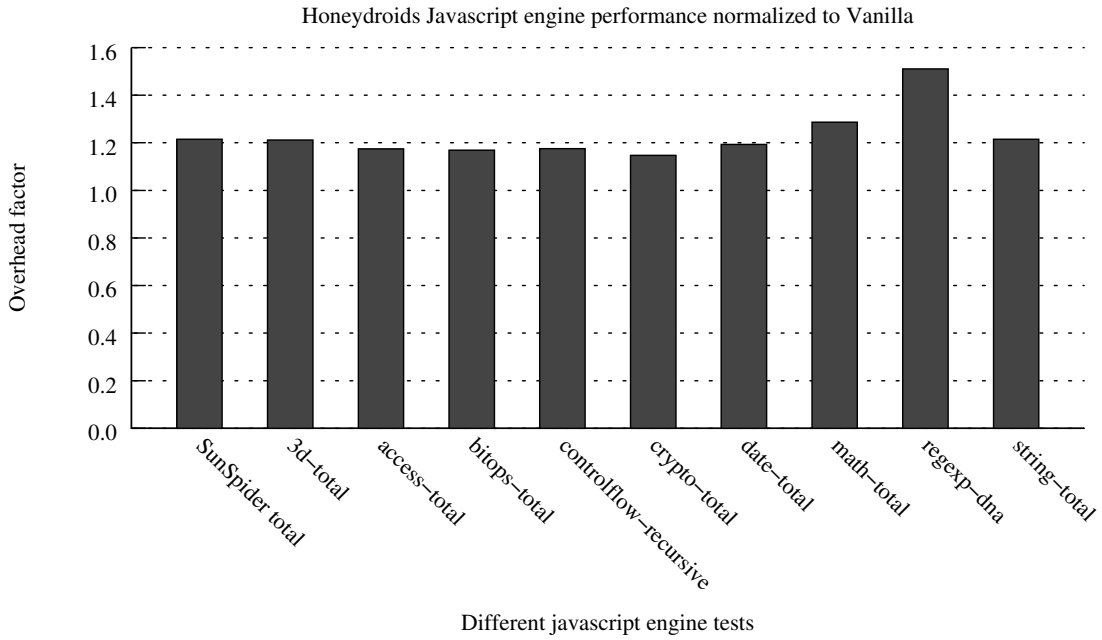

Figure 8.9: Vanilla and Honeydroid comparison for Sunspider

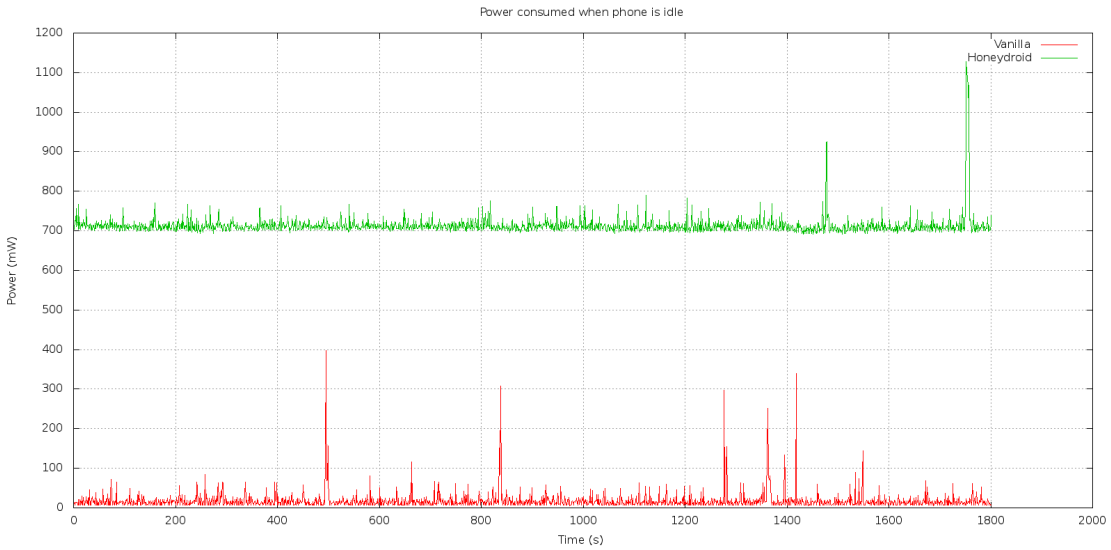

Figure 8.10: Idle test results for Vanilla and Honeydroid

Sunspider results necessitates further investigation as this overhead was not known or accounted for in previous research. Even though we cannot prove our speculations for this performance overhead in this thesis, we have introduced the matter to the scientific community.

8.6 Power Results

The power results for the two tests mentioned in 7.9 are reported in this section. The analyses of both tests are documented in one space due to their relatively similar context. For easy visualization and reporting we used a line curve and overlayed the results of Vanilla and Honeydroid on the same graph.

8.6.1 Idle state Results

The mean results for the idle state test are shown in Figure 8.10. The power consumed is reported in milli Watt(mW) while the time is reported in second(s). We can see that Honeydroid draws a minimum of at least 700mW of power while Vanilla consumes a minimum of 12mW. Honeydroid consumes a maximum of 1120mW towards the end of the test while Vanilla consumes a maximum of 400mW at the 500s mark.

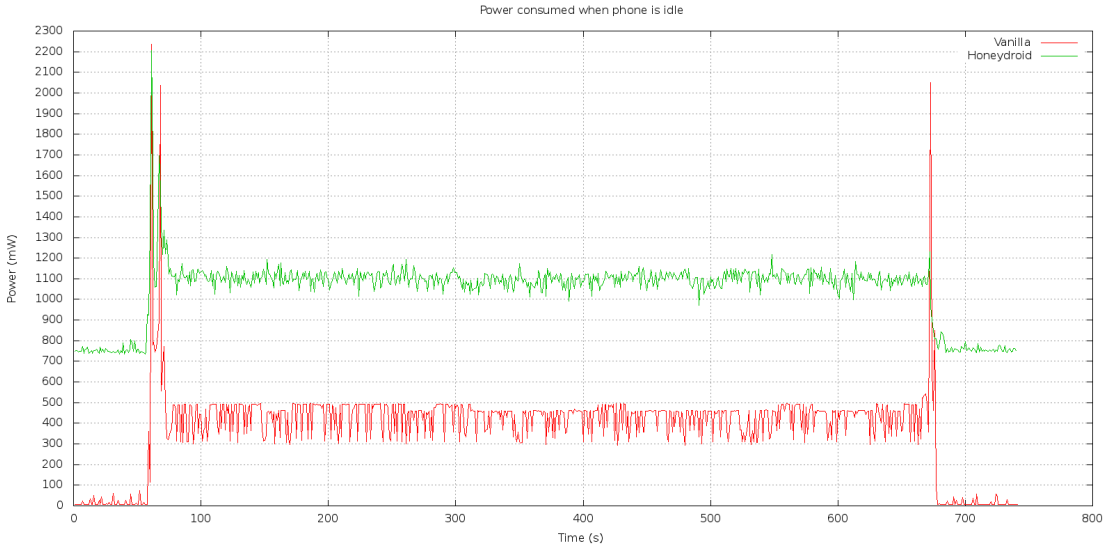

Figure 8.11: Voice call results

8.6.2 Voice Call Results

The mean results for the voice call test are shown in Figure 8.11. The power consumed is reported in milli Watt(mW) while the time is reported in second(s). We can see that both Vanilla and Honeydroid consume approximately the same amount of energy of 2200mW just when they receive the phone call at the 60s mark. During the call Vanilla consumes a maximum of 500mW and a minimum of 400mW while Honeydroid expends a maximum of 1100mW and a minimum of 100mW. When the call ends Vanilla peaks its power consumption to 2000mW while Honeydroid is more conservative by using only 1200mW.

8.6.3 Analysis

From Figures 8.10 and 8.11 it is very clear that Honeydroid consumes more power than Vanilla. This clearly means that the longevity of Honeydroids battery is very low compared with Vanilla. The reason for this increased power consumption by Honeydroid is due to the absence of power management. Android has an implemented and working power manager whereas L4Android does not. Power management in L4Android is a complicated affair due to the vertical and distributed structure of the Honeydroid op-

erating system where functionality and accounting are distributed across the working system.

The Honeydroid OS design is very modular, it is also a capability based system exercising access control. This modularity and security reduces transparency when it comes to power management. The Honeydroid compartment holds the granular information about the energy consumption of its applications but its incognizance of running in a virtualized environment and reduced privilege level makes it a bad power manager[51]. Therefore, the results from these tests do not account for the actual virtualization overhead in Honeydroid but instead they account for virtualization overhead plus the lack of power management. Improving the efficiency of para-virtualized platforms is a PhD thesis topic in itself and is beyond the scope of this work.

9 Conclusion

In this deliverable, we have described the design and implementation of Honeydroid: a virtualized smartphone honeypot. By instrumenting the entire smartphone operating system as a honeypot, and by employing a micro-kernel to enforce strong isolation, we realize a virtualized honeypot that provides us with key security properties: (1) Strong isolation between honeypot and monitoring VMs; (2) Integrity of logged data; (3) Enabler of attack-resistant, on-device malware containment mechanisms. This shows that, it is indeed possible to apply the classical honeypot concept to detecting intrusions into smartphone operating systems and that a virtualized honeypot framework such as the one presented in this report can be realized in a resource constrained environment like the present-day smartphone.

We show that features such as disk snapshots allow for interesting use-cases such as offline analysis of the disk trails that malware samples leave behind. We also demonstrate that a multiple VM setup allows us to perform rudimentary virtual machine introspection of which disk snapshots, system call histogram logging, and SMS filters presented in this report are good examples.

We present a methodology for evaluating our prototype with regard to performance, networking and battery overheads. Our evaluation results show a modest overhead for Android and Java benchmarks and minimal networking overheads. We also present results from tests to evaluate the battery life of the prototype. A systematic evaluation on the effectiveness of Honeydroid will be carried out as part of Deliverable 7.1.1 of the NEMESYS project.

Finally, we believe that Honeydroid is a significant step towards a systematic framework to detect attacks targeting smartphones. Honeydroid as a framework offers a platform to tease out additional monitoring tools. Furthermore, it enables research on the malware analysis front, that together with Honeydroid, advance the state-of-the-art.

Bibliography

[1] Android open source project. https://source.android.com/. Accessed 8th November, 2014.

[2] Anubis. https://anubis.iseclab.org/. Accessed 8th November, 2014.

[3] L4re. http://l4re.org/doc/index.html. Accessed 8th November, 2014.

[4] Network block device. http://nbd.sourceforge.net/. Accessed 8th November, 2014.

[5] Sebek. http://projects.honeynet.org/sebek/.

[6] *Hackbench man page from Ubuntu*, 1998. http://manpages.ubuntu.com/manpages/utopic/man8/hackbench.8.html.

[7] *LMbench home page*, 1998. http://www.bitmover.com/lmbench/.

[8] *Linpack for Java*, 2000. http://www.netlib.org/benchmark/linpackjava/.

[9] *Scimark2 for Java home page*, 2004. http://math.nist.gov/scimark2/.

[10] *0xbench for Android*, 2010. http://code.google.com/p/0xbench/.

[11] *Fannkuch problem description*, 2010. https://www.haskell.org/haskellwiki/Shootout/Fannkuch.

[12] *Nbody problem description*, 2010. https://www.haskell.org/haskellwiki/Shootout/Nbody.

[13] *Nbody problem description*, 2010. https://www.haskell.org/haskellwiki/Shootout/Nsieve.

[14] Open Source Honeypots: Learning with Honeyd. http://www.symantec.com/connect/articles/open-source-honeypots-learning-honeyd, November 2010.

[15] *Sunspider home page*, 2010. https://www.webkit.org/perf/sunspider/sunspider.html.

[16] *USRP Data sheet*, 2012. https://www.ettus.com/content/files/07495_Ettus_USRP1_DS_Flyer_HR.pdf.

[17] *GNU Radio Home page*, 2013. http://gnuradio.org/redmine/projects/gnuradio/wiki.

[18] *iptables man page*, 2013. http://ipset.netfilter.org/iptables.man.html.

[19] *Iperf 2 for Android*, 2014. http://magicandroidapps.com.

[20] *Iperf 2 website*, 2014. http://sourceforge.net/projects/iperf/.

[21] *RealPi app on Google Play*, 2014. https://play.google.com/store/apps/details?id=com.georgie.pi.

[22] *SPEC website*, 2014. http://www.spec.org/.

[23] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, 1969.

[24] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. *ACM SIGOPS Operating Systems Review*, 42(2):26–35, 2008.

[25] H.-J. Boehm. *GCBench for Java source code and comments*, 1999. http://hboehm.info/gc/gc_bench/applet/GCBench.java.

[26] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. *ACM SIGMETRICS Performance Evaluation Review*, 25(1):214–224, 1997.

[27] D. R. Cheriton. An experiment using registers for fast message-based interprocess communication. *ACM SIGOPS Operating Systems Review*, 18(4):12–20, 1984.

[28] L. Delosieres and A. Sanchez. Deliverable 2.3: Lightweight Malware Detector. Technical report, NEMESYS EU FP7 Project, 2014.

[29] K. M. Dixit. The spec benchmarks. *Parallel Computing*, 17, 1991. Benchmarking of high performance supercomputers.

[30] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, Dec. 2002.

[31] A. Fattori, K. Tam, S. J. Khan, A. Reina, and L. Cavallaro. CopperDroid: On the Reconstruction of Android Malware Behaviors. Technical Report MA-2014-01, Royal Holloway University of London, February 2014.

[32] Gartner Inc. Gartner Says Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments to Grow 4.2 Percent in 2014. `https://www.gartner.com/newsroom/id/2791017`. Date accessed, 8th November 2014.

[33] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. J. Magenheimer. Are virtual machine monitors microkernels done right? In *HotOS*, 2005.

[34] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM.

[35] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[36] A. Lackorzynski, J. Danisevskis, J. Nordholz, and M. Peter. Real-time performance of l4linux. In *Proceedings of the 13th Real-Time Linux Workshop*, 2011.

[37] S. Liebergeld, M. Lange, B. Shastry, B. Madalina, R. D'Alessandro, D. Carcía, and L. Delosières. Deliverable 2.1: Survey of Smart Mobile Platforms. Technical report, NEMESYS EU FP7 Project, 2014.

[38] J. Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[39] J. Liedtke. Improving ipc by kernel design. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 175–188. ACM, 1994.

[40] J. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, Dec. 1995.

[41] L. W. McVoy, C. Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[42] J. Mogul. *RFC 1191-Path MTU Discovery*, 1990. `http://tools.ietf.org/html/rfc1191`.

[43] NEMESYS Consortium. Deliverable 7.2.1: Analysis of attacks against the core mobile network infrastructure. Technical report, NEMESYS EU FP7 Project, 2014.

[44] J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[45] G. Portokalidis, A. Slowinska, and H. Bos. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 15–27, New York, NY, USA, 2006. ACM.

[46] N. Provos and T. Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Pearson Education, 2007.

[47] N. Quynh and Y. Takefuji. Towards an invisible honeypot monitoring system. In L. Batten and R. Safavi-Naini, editors, *Information Security and Privacy*, volume 4058 of *Lecture Notes in Computer Science*, pages 111–122. Springer Berlin Heidelberg, 2006.

[48] Range Networks. *OpenBTS Application Suite User Manual*, 2014. `http://wush.net/trac/rangepublic/raw-attachment/wiki/WikiStart/OpenBTS-4.0-Manual.pdf`.

[49] J. M. Rushby. Design and verification of secure systems. In *ACM SIGOPS Operating Systems Review*, volume 15, pages 12–21. ACM, 1981.

[50] C. Seifert, I. Welch, P. Komisarczuk, et al. Honeyc-the low-interaction client honeypot. *Proceedings of the 2007 NZCSRCS, Waikato University, Hamilton, New Zealand*, 2007.

[51] J. Stoess, C. Lang, and F. Bellosa. Energy management for hypervisor-based virtual machines. In *USENIX Annual Technical Conference*, pages 1–14, 2007.

[52] E. Vasilomanolakis, S. Karuppayah, M. Fischer, M. Mühlhäuser, M. Plasoianu, L. Pandikow, and W. Pfeiffer. This network is infected: Hostage - a low-interaction honeypot for mobile devices. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, pages 43–48, New York, NY, USA, 2013. ACM.

[53] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual hon-

eyfarm. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 148–162, New York, NY, USA, 2005. ACM.

[54] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, Bellevue, WA, 2012. USENIX.

[55] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.